22

22 A World of Interacting Objects

The two examples in this chapter illustrate the "world of interacting objects" that is typical of a program built using classes. They give you a practical model for an alternative to the "top-down" design approach used extensively in Part III. The examples also illustrate slightly simplified, informal versions of some of the schemes that are commonly used to document more elaborate object based programs.

Refcards example

The first example, "RefCards", is a little bit like the example in Section 17.3. In that example, a program manipulated "customer records" that contained data such as customer name, and amount ordered. Actually, that program manipulated a single structure in memory; the rest of the records were in a file. When a record was needed, it got loaded into memory. Now we can use things like an insane of a standard "list" class to hold a collection of records in memory, transferring these records to and from disk only when the program finishes and is restarted. The data records this time are "reference cards" – the sort of thing used to keep references to papers when you are doing scientific research. These records contain things like "authors' names", "paper title", "journal", and "page numbers". Although the RefCards program has some similarities to the earlier example, the use of classes in its design results in an implementation that is beginning to show a quite distinct structure.

Infostore example

The second example is an object-based reworking and elaboration of the "information retrieval" example from Section 18.3. That version of the program allowed the user to build up files containing newspaper articles, with an index based on a predefined set of keywords defined by an initialized data array. It used two programs. One added data to the file; the other performed single searches. Now we need something more general.

The program is to allow the user to define the "vocabulary" of keywords (as in the earlier example, it is actually a vocabulary of concepts as several different words can map onto a single concept used in the index). This vocabulary is to be extendible. If the user thinks of a new concept, or an additional word that maps onto an existing concept, then the system must allow this word to be added. This makes the system more flexible. It is not limited like the original to scientific articles. If you want a collection of travel articles, you simply make up a suitable vocabulary and start saving data.

The programs are also to be integrated. Vocabulary changes, article addition, and searches are all to be handled by the one program. The previous separate programs performed single functions – article addition, or search. The new program does not have a single function that can serve as a starting point for design. Instead, it gets built from objects – a "user interaction" object, a "vocabulary" object, and an "info store" object.

22.1 REFCARDS

Scholars doing research used to keep their records on "reference cards". Although computer data bases and file systems are now common, some people still use the old cards and many of the simpler computer systems use "cards" as a metaphor in their design. Reference cards would contain a number of data fields including: authors, title of paper, journal name, issue number, page number, year of publication, keywords, and possibly an abstract. Computer based versions have the advantage that you can easily search a collection of such "cards" checking each to identify those that contain a particular keyword, or a specified author.

Specification

The RefCard program will work with collections of "reference card" records. There is no specified maximum for the number of records in a collection, but the implementation can assume that the largest collections will have at most a few hundred records. A "reference card" record is to have text data fields for the author, title, journal, keyword, and abstract; it is also to have integer fields for issue number, year, first page, and last page. The text data fields should each hold 250 characters.

The "RefCard" program is to:

- allow a user to create a collection of "reference card" records.

- save a collection to a disk file and then reload a collection in subsequent runs of the program.

- let the user add a record to a collection, view an existing record, change a field in an existing record, or remove a record from the collection.

- let the user search for the first record with a particular word (name) in the "author", "title", "keyword", or "abstract" field; after displaying the first matched record the program is to allow searches for subsequent matching records. In addition to searches on specified individual fields, the search system should also allow for a search for a word in any of these fields.

- let the user display the contents of the "title", "keyword", "author", or "abstract" fields from all cards in the collection.

The program should be interactive, prompting the user for a command and for other data as needed. The command entry system should allow the user to view the range of commands appropriate in a given context.

22.1.1 Design

Preliminary design: Objects and their classes:

So, where do you start?

You start by saying "What objects are present in the running program?"

RefCard objects

Some objects are obvious. The program will have "RefCard" objects. These will own text strings (character arrays) in which data like author names, keywords, title and so forth are stored. They will also probably own some integer data like "year of publication". What do they do? They display their data, they can store their data to disk files, they can read data from disk files. They've got to get filled with data somehow; so it would seem reasonable if a "RefCard" could interact with a user allowing data to be entered or changed.

A list or dynamic array

The program is to work with at most a few hundred of these "RefCard" objects. They aren't that large (a little less than 1300 bytes), and since there will be only a few hundred at most, it is reasonable to keep them all in memory while the program is running (more than 750 records fit into a megabyte and that isn't much for most current PCs). Keeping all records in memory will make searches much faster. So, there will have to be something that holds the collection in memory. There is no "unique search key" that could be used for something like a binary tree. Instead we will need something like a "list" or "dynamic array". A dynamic array seems more appropriate. Removal operations are going to be rare; lists are better than dynamic arrays only in cases where removals are frequent. The dynamic array can be an instance of an "off the shelf" class from a class library.

CardCollection object

Although a dynamic array can be used to hold the actual collection, there had better be something a little more elaborate. We need operations that involve the collection as a whole – like the searches (as well as operations like saving the collection to a disk file). The search system needs information like the string that is sought, and the current position in the collection if we are doing operations like "find" and "find again". We will also need some idea like "current position" if we are to handle requests like "show card ...", and "remove card ...". A "CardCollection" object could own the actual dynamic array with the cards and, in addition, other information like search string, position, and name of the file. A `CardCollection` object could organize searches and similar operations like viewing a particular field from all records. What exactly does a `CardCollection` own and do? Don't know. Its role will become clearer as we iterate through the design process.

UserInteraction object

Although the `CardCollection` object could handle some of the interactions with the user (like getting information needed for a search), it will be useful to have more general interactions handled by a "UserInteraction" object. Actually, this is probably not strictly necessary for this program. However, as we move to more elaborate programs (particularly those built on top of framework class libraries as discussed in Part V) we will see "standard" arrangements of classes. It will be "standard" to have some "UserInteraction" object that controls the overall program flow. This `UserInteraction` object will create the `CardCollection` object, get the top level commands from the user ("add a card", "show a card", "change a card", "do a search", …), sort out files and do similar tasks. In this program, the `UserInteraction` object will only have one `CardCollection`; but if a `UserInteraction` object could use more than one window for input and output it could have a separate `CardCollection` associated with each separate window (that is how word processors and other programs typically work).

main()?

If the `CardCollection` object is organizing everything, what is there left for the "main program"? Not much. A main program for one of these object based systems is usually simple: create the principal object, do any other initializations, tell principal object to "run", when "run" finishes do any tidying up that may be needed. In fact, we can write the main program already:

```
int main()
{
    UserInteraction      UI;
    UI.Initialize();
    UI.Run();
    UI.Terminate();
    return 0;
}
```

Classes and objects

We aren't yet into "class hierarchies" (see next chapter). When we get class hierarchies we will have to do a bit more work characterizing classes and finding relationships between classes. For now, things are simple. The various types of object identified so far correspond directly to the classes that we will need. They are summarized in Figure 22.1. Most are shown as "fuzzy blobs". The fuzzy blobs represent "analysis" level classes. We have a rough idea as to what they own and what they do, but we can't yet define any hard boundaries.

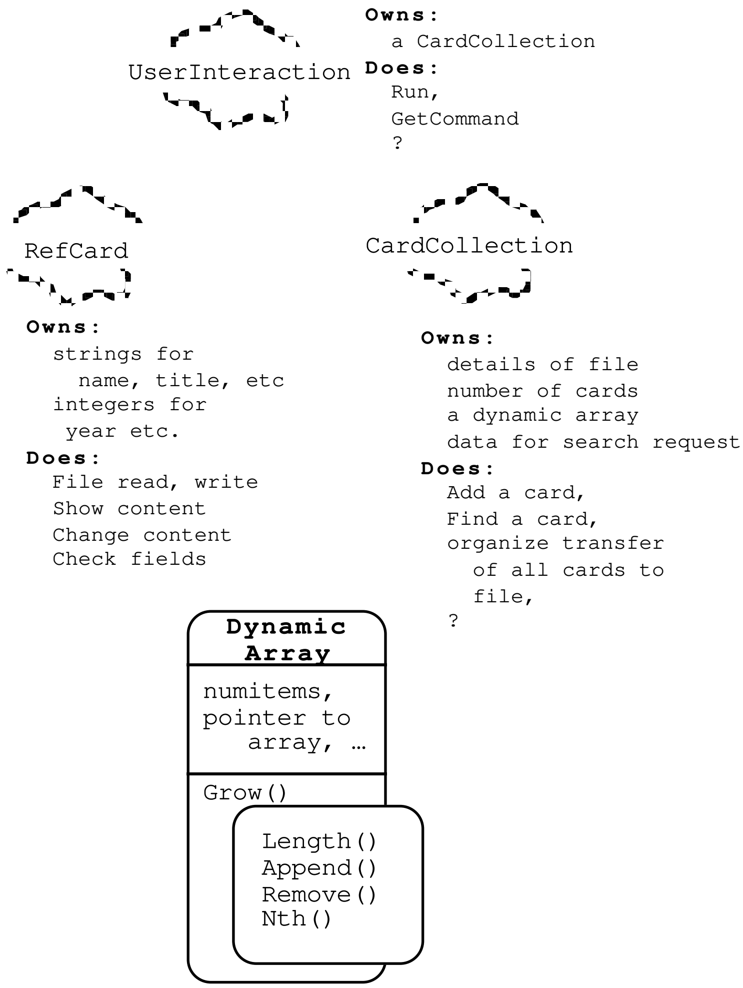

Figure 22.1 First idea for classes for RefCards example.

Class `DynamicArray` is an exception. It is shown as a design level class diagram with a firm boundary, exact specification of its interface and private data. After all, its from a class library and has already been designed and implemented. Our next task will be to firm up those fuzzy blobs so that they too can be defined with firm boundaries.

Design 2: Characterize interactions among objects

The next stage of the design process is typically iterative. Our aim in this stage is to clarify what instances of these classes own and exactly what they do. We aren't yet interested in exactly how they do their tasks, just trying to identify how the overall work of the program gets split up into tasks that each different object must perform.

Scenarios

The best approach to assigning responsibilities is to try out "scenarios" that illustrate how some of the program's major tasks can be accomplished. A simple scenario for this program would be "the things that happen when its time to terminate"; it will be simple because the only important thing that will happen is that the `CardCollection` must save all the cards back to a disk file. A more complex scenario would be "the things that happen when we search for cards with the word 'Stroustrup' in their 'author' fields". This scenario would be more complex because it probably involves more types of object and more elaborate processing tasks.

While working through these scenarios you guess. You guess things like "object-1 will ask object -2 to perform task A". Then, you examine the implications. If object-2 is to perform task A then it had better own (or at least have access to) all the data needed to carry out this task. If this scenario has object-2 owning some data, then those data had better not appear as belonging to a different object in some other scenario. If object-2 doesn't own the data but has "access to them", then you had better sort out how object-2 got that access. Presumably some other object gave object-2 a pointer to the data, but which other object and when? You guess again noting down an extra responsibility for another object (and, hence, a function for its class).

Naturally, some of your guesses are wrong. You run through a few scenarios. Discover that in different scenarios you've allocated ownership of some specific data to different objects. You have to decide which class of object really should own the data and go back and change the scenario that is incorrect.

Of course, this is still a fairly simple program so the scenario analysis and other mechanisms for "fleshing out" the roles of the classes will be completed rather easily.

<u>Example scenario: initialization</u>

Assumptions: 1) the main program has already created a `UserInteraction` object, 2) the constructor for the `UserInteraction` object created a `CardCollection` object, 3) this `CardCollection` object is "empty", it will have a dynamic array with some default number of slots, and it will have initialized things like its count of cards to zero.

Initialization task: get the user to enter a filename, if the file exists read data on existing cards, otherwise create an empty file where a new collection of cards can be saved.

Possible interactions:

1. `UserInteraction` object gets user to enter a filename. It could open the file, but handling the file might be better left to the `CardCollection` object. So ...

2. `UserInteraction` object asks `CardCollection` object to open the file, passing the name of the file as an argument in the request.

3. The `CardCollection` object should first try to find an existing file (an "open" operation with "no-create" specified).

   If this works then existing data should be loaded (see next step).

   If that operation failed, the `CardCollection` object should try to create a new file.

   If it can open a new file, the `CardCollection` object should report success (it should also make certain that all its data fields are initialized properly, though possibly this should already have been done in its constructor).

4. If the `CardCollection` object was able to open an old file, it has to load the existing cards into memory.

   It is going to have to have a loop in which it creates `RefCard` objects, gets them filled with data from the file, and then adds them to its `DynamicArray` (lets call that `fStore`).

   It will be easiest if the first data item in the file is an integer specifying how many cards there are. The `CardCollection` object can read this integer into its card count (`fNumCards`) and then have a loop like the following:

```
for(int i = 1; i<= fNumCards; i++) {
        RefCard *r= new RefCard;
        r -> ReadFrom(input file);
        fStore.Append(r);
        }
```

   The `CardCollection` object doesn't know what data is in a `RefCard` so it can't read the data. A `RefCard` does know what data it wants. So, as shown, the `CardCollection` object asks each newly created `RefCard` to read itself.

   When this process finishes, we will have a set of `RefCards` that have each been created in the heap. Their addresses will be held in the `DynamicArray fStore`. Because it owns `fStore`, the `CardCollection` object can get at the individual `RefCards` whenever it needs them.

   There should be some checks for successful file transfers. If everything ran OK, then the `Load` function should return a success indicator to the `Open` function which can the report success to the `UserInteraction` object.

5. The `UserInteraction` object should check the success/failure indicator returned by the `CardCollection` object; if the file couldn't be opened the `UserInteraction` object should either terminate the program or loop to allow the user to guess another file name.

6. Finally, the `UserInteraction` object should tell the `CardCollection` object to print a status report so that the user knows how many cards are in the collection.

Diagramming the object interactions of a scenario

It is usually helpful to represent such interactions through a diagram, like that shown in Figure 22.2. The diagram shows things that happen at different times; the time increases as you go down the diagram. The entries shown across the diagram illustrate what different objects are doing.

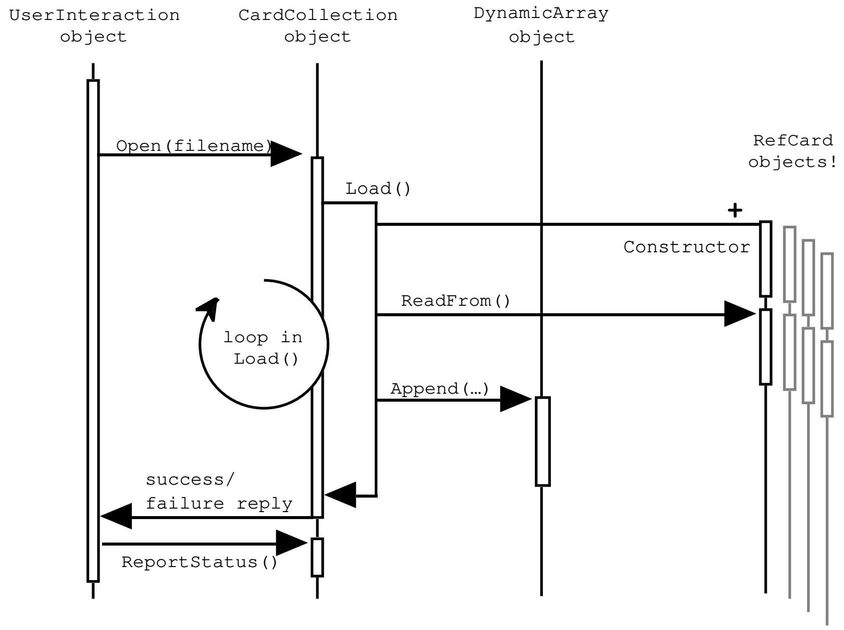

Figure 22.2 Object interactions when loading from file (activities resulting from execution of UserInteraction::Initialize() for an UI object).

Back in Chapter 15 where design techniques for top down functional decomposition were reviewed, diagrams were rated as less useful than text descriptions and pseudo code. It was suggested that diagrams like those charting possible function calls weren't that informative. These class interaction diagrams do provide a much clearer indication of the dynamics of a program. They can capture what is going on in each significant subtask that must be performed.

Meanings of symbols in diagram

A diagram like that shown in 22.2 doesn't try to represent everything; for example, there is nothing about the file not opening and having to be created. The focus is on the more important interactions between objects. The classes of the objects involved are indicated by the class labels across the top. Single vertical lines show where an object is in existence. The `UserInteraction` (UI), `CardCollection`, and `DynamicArray` object all exist before the start of the diagrammed scenario and continue to exist after its finish. The `RefCard` objects only get created part way through the scenario, so their vertical lines start half way down.

The outline rectangles indicate where an object is active, i.e. executing a function or its own function has invoked a global function or action by some other object. In the example, all activity is part of `UserInteraction::Initialize()` (the long rectangle for the `UserInteraction` object). Arrows indicate function calls (and sometimes are used to provide information on results returned). The line with the '+' tag indicates a place where a new object is to be created.

Equivalence between diagram and earlier text description

Thus, the diagram shows the process of a call from the `Initialize()` function of a UI object to the `Open()` function of a `CardCollection` object. This `Open()` function calls `Load()` (executed by the same `CardCollection` object). Function `CardCollection::Load()` has a loop. In this loop, a `RefCard` object gets created (the '+' line), then gets asked to execute its `ReadFrom()` function. Then the `DynamicArray` object gets asked to do an `Append()` operation.

Results:

New responsibilities identified from scenario

The process of analysing and diagramming this scenario has added new functions to the responsibilities proposed for class `CardCollection`. It is going to have to have functions like:

```
int CardCollection::Open(char*)         open file with given name
void CardCollection::Load()             create RefCards, get
                                        them filled in with data
                                        from file
void CardCollection::ReportStatus()     state what was read
```

The `CardCollection` object should probably be responsible for recording the name of its file (this name will be needed in some of the output shown to the user, like that from `ReportStatus()`). It had also better have an `fstream` data member so that it can actually keep hold of the file from the time its told to `Open()` till the time its told to `Close()` and save the data. So, we have also got a couple of extra data members:

```
CardCollection {
public:
    …
private:
    int     fNumCards;
    char    *fFileName;
    fstream       fFile;
    DynamicArray  fStore;
    …
};
```

Example scenario: termination

Assumptions: The `CardCollection` object has an open `fstream` to which it can write its cards.

Termination: The termination stage of the program requires that all `RefCards` get saved to file and then they should probably be deleted (not absolutely necessary here as the program is about to finish, but it would matter if the program was supposed to allow the user to continue by opening another card collection).

Possible Interactions:

Interactions on termination

Figure 22.3 diagrams an idea as to the interactions. The `UserInteraction` object will ask the `CardCollection` object to `Close()`. The `CardCollection` object would start by calling its own `Save()` function. This would start by setting the file so that any existing data gets overwritten, then it would write out the number of cards, after this there would be a loop in which each `RefCard` object in the `DynamicArray` gets told to save its own data. The code would be something like:

```
fFile.seekp(0);
fFile << fNumCards << endl;
for(int i=1; i <= fNumCards; i++) {
    RefCard *r = (RefCard*) fStore.Nth(i);
    r->WriteTo(fFile);
    }
```

Once the cards had been saved, the file should be closed and then there could be a second loop in which they get removed from the `DynamicArray` and are then deleted. This would be done in the `CardCollection::Close()` function:

```
…
fFile.close();
for(int i = fNumCards; i > 0; i--) {
    RefCard *r = (RefCard*) fStore.Remove(i);
    delete r;
    }
```

When the function `CardCollection::Close()` was completed, control would return to `UserInteraction::Terminate()`. There would then be an opportunity to delete the `CardCollection` object.

Results:

New responsibilities identified

This analysis identifies just one extra function. Class `CardCollection` will have to define a `Save()` routine; like `Load()` this will be a private member function.

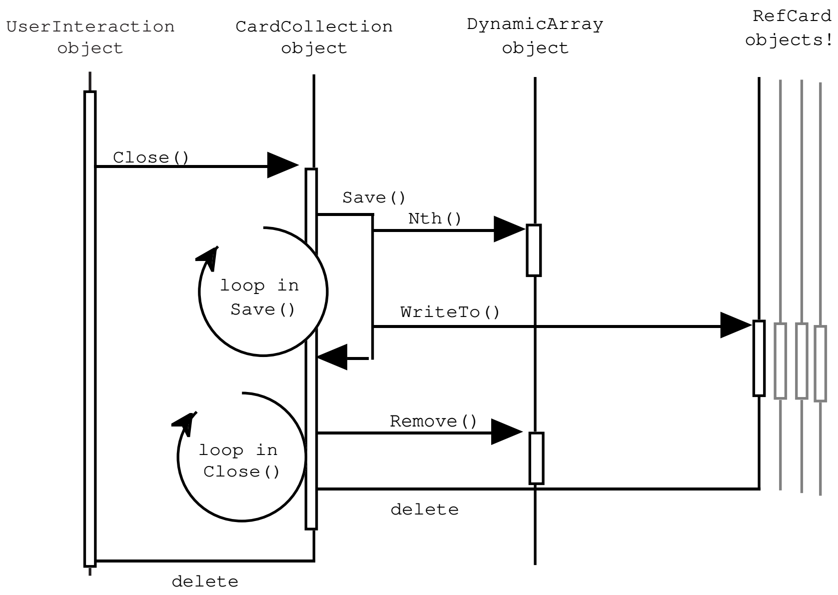

Figure 22.3 Object interactions resulting from UserInteraction::Terminate().

Example: Interaction between the User and the UserInteraction object

The code for handling interactions between the human user and the UI object will itself be simple. This code will form the body of the function `UserInteraction::Run()`. Basically, all we need is a loop that gets a command from the user (a single letter will do), the command will (usually) be easy to convert into a request that the `CardCollection` perform some action like showing an existing card, or adding a new card.

A user command like "add a card" can be passed directly to the `CardCollection` object. A command like "change contents of a card" or "delete a card" will require additional input to identify the card. (Cards may as well be identified by their sequence number in the collection; these sequence numbers can appear in listing so that the user knows which card is which.) We will need a few simple auxiliary routines to get extra input data.

`UserInteraction::Run()` will be something along the following lines:

Simple command loop

```
void UserInteraction::Run()
{
    int     done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(; !done; ) {
```

```
             char command = GetCommand();
             switch(command) {
case 'q' :         done = 1; break;
case '?' :         Help(); break;
case 'a' :         fCollection->AddCard(); break;
case 'c' :         DoChange(); break;
case 'd' :         DoDelete(); break;
…
…
case 'v' :         fCollection->DoView(); break;
default :
                   cout << "Command " << command <<
                          " not recognized" << endl;
                   }
             }
}
```

Auxiliary private member functions

It implies the existence of a largish number of very simple auxiliary routines. These will all become additional private member functions of class `UserInteraction`. A function like `GetCommand()` will just read a character, convert it to lower case, and return it.

Functions like `DoChange()` and `DoDelete()` both need the user to input a card number. Obviously the number entered has to be validated; it will have to be in the range 1 to n where n is the number of cards owned by the collection. (Note, the `UserInteraction` object has to be able to ask the `CardCollection` object how many cards it has.) Since this number input routine is needed by several routines, it might as well become another private member function. Once these "DoX()" functions have got any necessary additional input, they will call matching functions of the `CardCollection` object.

Elaboration of these simple member functions can be handled by using much the same sort of top down functional decomposition techniques as presented in Part III. Here the "top" is a single (moderately complex) member function of a specific class.

Results:

New responsibilities identified

Class `CardCollection` must report the number of cards it has, so we need:

```
int CardCollection::NumCards()
```

as an extra public member function. Class `CardCollection` will also need `AddCard()`, `DoView()`, `ShowCard(int cardnum)` and similar functions. Some additional scenarios will be needed to clarify the prototypes (argument lists) for these functions.

A largish number of simple private member functions have been identified for class `UserInteraction`. We are going to need:

```
char    UserInteraction::GetCommand();     get character
```

```
int      UserInteraction::PickACard();        get valid card number
void     UserInteraction::Help();             list valid commands
void     UserInteraction::DoDelete();         organize delete
void     UserInteraction::DoChange();         organize change
void     UserInteraction::DoShow();           organize show
```

Example scenario: add a card

Adding a card: The collection will have to create a new `RefCard`, get the `RefCard` to interact with the user to obtain the information that it needs for its data members, and then it will have to add the card to the dynamic array. An interaction diagram is shown in Figure 22.4.

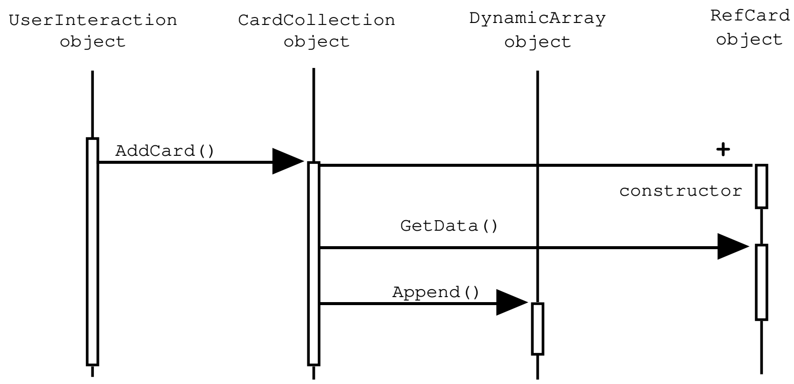

Figure 22.4 Object interactions resulting from a User's "add card" command.

Most of the work can be left to the newly created `RefCard` object. This can prompt the user to enter data for each of the data members (title, authors, year of publication etc.).

Results:

Need functions:

```
void     CardCollection::AddCard();
```

and

```
void     RefCard::GetData();
```

Example scenario: getting a card shown or changing an existing card

These two operations are going to involve very similar interactions among the participating objects. The pattern is illustrated in Figure 22.5.

The `UserInteraction` object will execute a routine (`PickACard()`) that prompts the user for the card number. This will involve an interaction with the `CardCollection` object to make sure that the number entered is in range.

If a valid card number is entered, the `UserInteraction` object will ask the `CardCollection` object to `ChangeCard()` or `ShowCard()`.

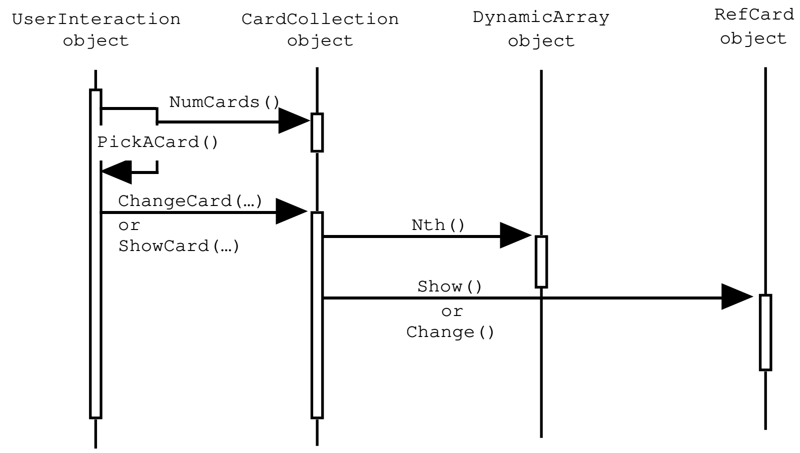

Figure 22.5 Object interactions resulting from a User's "show card" or "change card" commands.

The `CardCollection` object will use the `Nth()` member function of its `DynamicArray` to get a pointer to the chosen `RefCard`. The "show" or "change" command will then be forwarded to the `RefCard` object. A `RefCard` will handle "show" by displaying the contents of all data members. A "change" command will involve prompting the user to identify the data member to be changed (single character input), display of current contents of that data member, and acceptance of new input value.

Example scenario: deleting a card

The interactions for a delete command are summarized in Figure 22.6

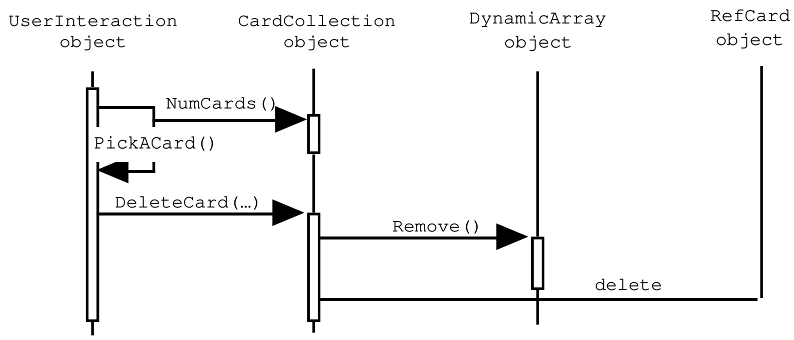

Figure 22.6 Object interactions resulting from a User's "delete card" command.

Example scenario: viewing a field from every card

View: The user is prompted to identify which field is to be viewed (choice restricted to 'author', 'title', 'keywords', or 'abstract'). The contents of the chosen field should then be displayed (along with a card identifier number) for each card in the collection.

Possible Interactions:

Figure 22.7 diagrams an idea as to the interactions. The `UserInteraction` object can simply pass a "view" request to the `CardCollection` object.

The `CardCollection` object will first have to get the user to identify the field that is to be displayed. This will involve the use of another auxiliary private member function, `GetField()`. This function will prompt the user to chose among the allowed fields (again, a single letter code should suffice for input). The function should return an integer identifier.

These "field identifiers" are shared with the `RefCard` objects. It would probably be best if they were defined in the RefCard.h header file.

Loop processing each card in the collection

After the code to get a field identifier, the `DoView()` function will need a loop in which it accesses each `RefCard` from its collection and tells it to print the contents of the chosen field. The code will be along the following lines:

```
for(int i = 1; i <= fNumCards; i++) {
    cout << i << "\t";
    RefCard *r = (RefCard*) fStore.Nth(i);
    r->PrintField(field);
    cout << endl;
    }
```

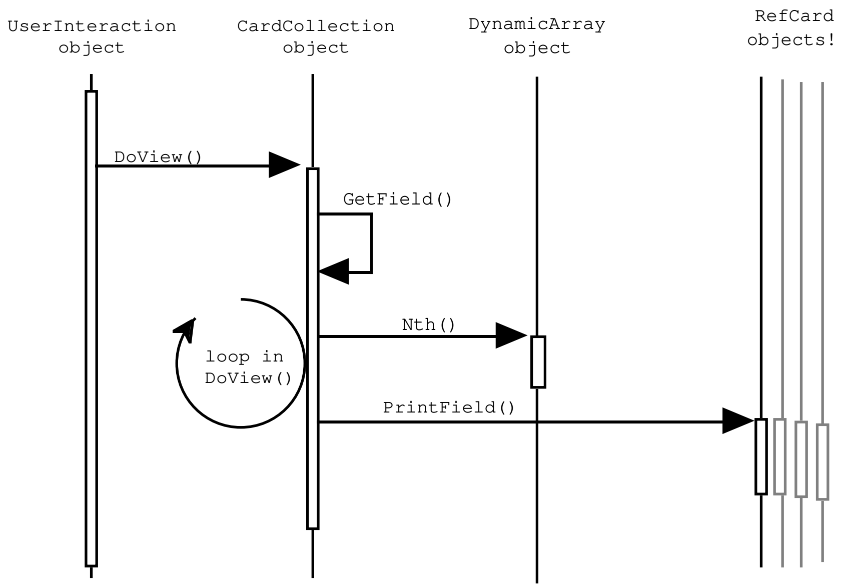

Figure 22.7 Object interactions resulting from a User's "view" command.

Results from scenario:

New functions: `CardCollection::DoView()`, `CardCollection::GetField()`, and `RefCard::PrintField()`.

Example: Searching for cards with a given word

The actual interactions among objects when doing a simple search would be somewhat similar to those shown for "view".

Simple form of search

Once again, the `CardCollection` object would need to get the user to identify the field (data member) of interest; though there is a change here in that system is supposed to allow a search on "any" of the four fields in addition to searches on individual fields. (Probably, function `GetField()` should be extended so that it has a parameter that indicates whether "any" is an acceptable input. This parameter would be "false" if `GetField()` were being called to find a field for "view" but "true" in the case of a search.)

After identifying the field of interest, the `CardCollection` object would have to get the user to enter the string (the name or word that is to be found in the search). It could then have a loop asking each `RefCard` in turn to "check a field" (rather than print a field as in "view').

Checking a field for a string

Coding the search for a word in a character array like a title is not a problem. There is a function in the string library that does exactly this (function `strstr()`).

More complex search required by specification

The main problem is that a simple search would list all the cards that matched. The specification required something like the "Find…" and "Find Again…" commands that you get with most word processors.

We will have to have two functions:

```
CardCollection::DoFind()
CardCollection::DoFindAgain()
```

DoFind()

The `DoFind()` function will do the more elaborate work. It will prompt the user to identify the field and the string, and organize a search for the first card that matches. It will have to arrange for the `CardCollection` object to store state information defining the field, string, and position in the collection where the first matching card was found.

DoFindAgain()

The `DoFindAgain()` function should check that there is a search in progress (string defined, position reached last time set etc). If that is OK, it should have a loop that works through successive cards until another match is reached. It should then update the state data so that another later call will continue the search.

This implies that we need some additional data members in class `CardCollection`:

```
char        *fFindString;
int         fFindPos;
int         fSearchField;
```

These are going to have to be set appropriately in the constructor.

Finalising the design

The design process using scenarios (and supplementary top-down functional decomposition when you get a complex member function) have to continue until you fully can characterize your classes. You need to get to the point where you can write down a complete class declaration and provide short (one sentence) descriptions of each of the member functions.

For this example, we eventually get to the following:

Class RefCard declaration

```
class RefCard {
public:
    RefCard();
    /*
    Disk i/o
    */
    void   ReadFrom(fstream& s);
    void   WriteTo(fstream& s) const;
    /*
    Communication with user
```

```
    */
    void    Show() const;
    void    GetData();
    void    Change();
    int     CheckField(int fieldnum, char *content);
    void    PrintField(int fieldnum);
private:
    char    fAuthors[kNAMEFIELDSIZE];
    char    fTitle[kNAMEFIELDSIZE];
    char    fJournal[kNAMEFIELDSIZE];
    char    fKeywords[kNAMEFIELDSIZE];
    char    fAbstract[kNAMEFIELDSIZE];
    short   fFirstPage;
    short   fLastPage;
    short   fIssue;
    short   fYear;
};
```

As required by the specification, the character arrays used to store titles are fixed sized.

Class RefCard, function specifications

Constructor
Initialize all character arrays to blank string, all integers to zero.

ReadFrom, WriteTo
Read data from (write data to) a text file.

Show
Print identifying field labels and contents of all data members.

GetData
Prompt user and then read values for each data member in turn.

Change
Get user to identify data member to be changed (enter a letter), output details of current contents of that data member, read replacement data.

CheckField
Integer argument identifies data member to be checked, string argument is content to be searched for using `strstr()` function from string library.

PrintField
Integer argument identifies data member that is to be output.

Class CardCollection declaration

```
class CardCollection {
public:
    CardCollection();
    /*
    Attaching to file
    */
    int     Open(const char filename[]);
    void    Close();
    /*
    Main commands
    */
    int     NumCards() const;
    void    ReportStatus() const;
    void    AddCard();
    void    DeleteCard(int cardnum);
    void    ShowCard(int cardnum) const;
    void    ChangeCard(int cardnum);
    void    DoFind();
    void    DoFindAgain();
    void    DoView();
private:
    int     GetField(int anyallowed);
    void    Load();
    void    Save();
    int     fNumCards;
    char    *fFileName;
    fstream       fFile;
    DynamicArray fStore;
    char    *fFindString;
    int     fFindPos;
    int     fSearchField;

};
```

Class CardCollection function specifications

Constructor

Initialize data fields, `fNumCards` is zero, `fFindString` is `NULL` etc.

Open

Either open existing file and call `Load()`, or create a new file. If can't do either report error.

Close

Call `Save()` to get cards to file, then clean up deleting cards.

NumCards

Report number of cards in current collection.

ReportStatus

Print out name of file, and details of number of cards.

AddCard

Create a card, get it to obtain its data from the user, add to collection, update count of cards owned..

DeleteCard

Remove identified card from collection then delete it, update count of cards owned.

ShowCard

Get pointer to chosen card from dynamic array, tell card to show itself.

ChangeCard

Get pointer to chosen card from dynamic array, tell card to interact with user to get changes.

DoFind

Use `GetField()` (specifying any as OK) to allow user to pick field, then prompt for string, then loop through cards asking them to check the specified field-string combination. Stop as soon as get a match, asking the card to show itself and recording, in state data, the point where match was found. If no matches, warn user and discard the search information

DoFindAgain

Check that there is a search in progress. If not, warn user. If there is a search, continue from current point in collection checking cards until either find a match or there are no more cards.

GetField

Use simple "menu" selection system to let user pick a field.

Load

Read number of cards in file, then loop creating cards, letting them read their data from file, and adding them to collection.

Save

Write number of cards to file, then let each card write itself to file.

Class UserInteraction declaration

```
class UserInteraction {
public:
    UserInteraction();
```

```
        void   Initialize();
        void   Run();
        void   Terminate();

    private:
        char   GetCommand();
        int    PickACard();
        void   Help();
        void   DoDelete();
        void   DoChange();
        void   DoShow();

        CardCollection *fCollection;
    };
```

Class UserInteraction function specifications

Constructor
Create a `CardCollection` object

Initialize
Ask the user to enter a filename, tell the `CardCollection` object to try to open that file.

Run
Loop getting and processing user commands until a "quit" command is entered.

Terminate
Tell the `CardCollection` object to close up, then get rid of it.

GetCommand
Get single character command from user.

PickACard
Prompt user for a card number, make certain that it is in range (ask `CardCollection` object how many cards there are to chose from).

Help
Print explanation of available commands.

DoDelete, DoShow, DoChange
Use `PickACard()` to get the card number then call the corresponding member function of the `CardCollection` object.

As well as developing the class declarations and function summaries, you might want to produce "design diagrams" for the classes like that shown in Figure 19.1.

File (module) structure of program

Another design choice you now have to make is how these components will be organized in files. Here it would be appropriate to have the files:

CC.h, CC.cp	CardCollection class
D.h, D.cp	the dynamic array
main.cp	the little main() driver routine
RefCard.h, RefCard.cp	RefCard class
UI.h, UI.cp	UserInteraction class

"Header dependencies"

Because the objects of these classes interact so much, there are lots of inter-dependencies in the code. For example, when compiling the code in UserInteraction.cp, the compiler has to be able to check that all those requests to the `CardCollection` object are valid. This means that it will have to have read the CardCollection.h file before it compiles UserInteraction.cp.

Direct (uses) dependencies

It is often useful to draw up a diagram showing the interrelationships between files so that you remember to #include the necessary headers. Figure 22.8 illustrates some of the relations for this program.

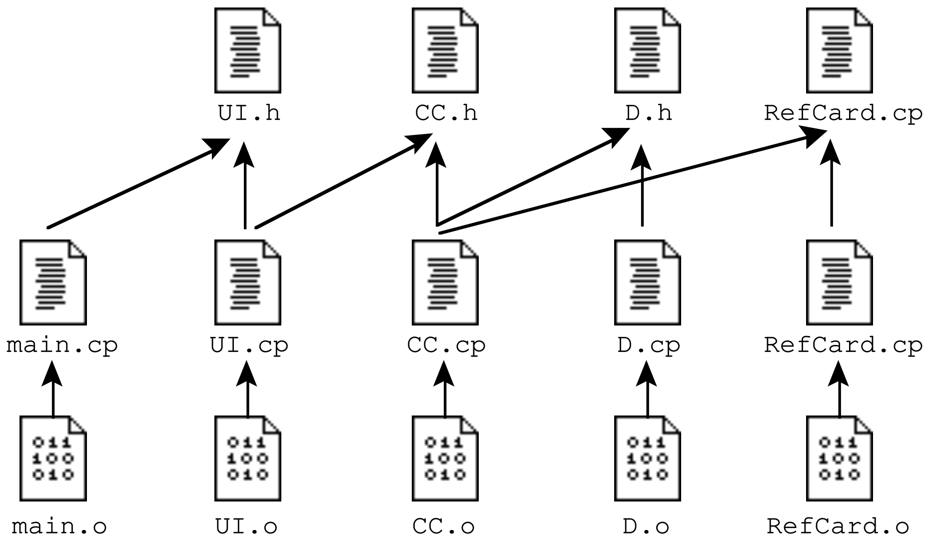

Figure 22.8 Illustration of direct file dependencies in the RefCard program.

The code in CardCollection.cp uses `RefCard` functions and `DynamicArray` functions so it must #include both these header files. Class `UserInteraction` needs the declaration of class `CardCollection` so it must include CC.h; similarly main needs #include UI.h.

Additional (indirect) dependencies

There are further dependencies. Although `UserInteraction` doesn't make any direct use of a `DynamicArray` it does need to know about this class. Similarly, the main program needs to know about the existence of class `CardCollection`. These dependencies result from the presence of data members in classes.

Class `CardCollection` has a `DynamicArray` as a data member. When the compiler is trying to process the code in UI.cp it will need to work out the size of a CardCollection object and so will need to have already read the file D.h.

When processing main.cp, the compiler would need to be reassured that the `CardCollection*` pointer data member in class `UserInteraction` was referring to a defined type of structure. It wouldn't need to read the complete declaration of class but would need to have the class declared (i.e. the UI.h file would need to include a declaration like "`class CardCollection;`" as well as the full declaration of class `UserInteraction`).

Compiler error messages relating to missing headers are often obscure

You have to sort out all the additional indirect dependencies so that you can #include all required files. If you forget some, you will get numerous compiler error messages. The actual messages may be obscure – complaints like "Illegal cast from int to void*", or "Undefined function" for a function that you know is defined. Compilers can get quite confused when they encounter references to classes that they haven't seen declared. If you get such odd compiler errors, start by checking that you included the right header files and in the correct order (as `CardCollection` uses `DynamicArray`, the #include "D.h" should come before the #include "CC.h" to make certain that class `DynamicArray` has been declared before an instance of the class gets used).

22.1.2 Implementation

As always, if the design is complete then the implementation is trivial.. The main program has already been given.

UserInteraction

The header file UI.h would contain the class `UserInteraction` declaration shown earlier. The presence of the `CardCollection*` data member means that the header file would have to contain a declaration the existence of class `CardCollection`. The implementation file, UI.cp, would need to #include stdlib, ctype, UI, and CC.

The constructor simply creates the `CardCollection`. The `Initialize()` member function will get a filename from the user and ask the `CardCollection` to open the file; if this fails the program terminates (this call to `exit()` lead to the need to #include stdlib.h). Function `Terminate()` gets the `CardCollection` to close up before it is deleted.

constructor

```
UserInteraction::UserInteraction()
```

```
{
    fCollection = new CardCollection;
}
```

Initialization, getting filename

```
void UserInteraction::Initialize()
{
    char   buff[100];
    cout << "Enter name of file with cards : ";
    cin >> buff;
    int status = fCollection->Open(buff);
    if(status < 0) {
           cout << "Sorry, can't open (or create) file."
                  "Giving up" << endl;
           exit(1);
           }
    fCollection->ReportStatus();
}

void UserInteraction::Terminate()
{
    fCollection->Close();
    delete fCollection;
}
```

The highlighted statements are typical of the "hey object, do action" calls that pervade the code.

The complete version of the `Run()` member function is:

Interaction loop for user commands

```
void UserInteraction::Run()
{
    int     done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(; !done; ) {
           char command = GetCommand();
           switch(command) {
case 'q' :        done = 1; break;
case '?' :        Help(); break;
case 'a' :        fCollection->AddCard(); break;
case 'c' :        DoChange(); break;
case 'd' :        DoDelete(); break;
case 's' :        DoShow(); break;
case 'f' :        fCollection->DoFind(); break;
case 'g' :        fCollection->DoFindAgain(); break;
case 'v' :        fCollection->DoView(); break;
default :
                  cout << "Command " << command <<
                         " not recognized" << endl;
                  }
           }
}
```

Menu based programs like this should try to include a help function that explains their options:

Built in help

```
void UserInteraction::Help()
{
    cout << "Commands are : " << endl;
    cout << "\ta  Add a new card." << endl;
    …
    …
    cout << "\tv  View one field from all cards" << endl;
    cout << "\tq  Quit" << endl;
}
```

Functions `DoDelete()`, `DoChange()`, and `DoShow()` are very similar; `DoDelete()` can represent them all:

DoDelete() and similar functions

```
void UserInteraction::DoDelete()
{
    int which = PickACard();
    if(which < 1)
           return;
    fCollection->DeleteCard(which);
}
```

They all use the auxiliary function `PickACard()` to get a valid card number:

Auxiliary input functions

```
int UserInteraction::PickACard()
{
    if(fCollection->NumCards() < 1) {
           cout << "There aren't any cards so you can't do"
                          "that now." << endl;
           return 0;
           }
    cout << "Which card? (Enter number in range 1 to " <<
           fCollection->NumCards() << ") : " << endl;
    int aNum;
    cin >> aNum;
    if(!cin.good()) {
           cout << "??";
           cin.clear();
           cin.ignore(100, '\n');
           return 0;
           }

    if((aNum < 1) || (aNum > fCollection->NumCards())) {
           cout << "Invalid input, ignored." << endl;
           return 0;
           }
    return aNum;
}
```

The other auxiliary input function, `GetCommand()`, is used by `Run()` to get a single input character. The call to `ignore()` removes ("flushes") any other input remaining in the stream (this avoids problems when a user does something like type a command name, e.g. 'view,' instead of just a command letter 'v').

```
char UserInteraction::GetCommand()
{
    char ch;
    cin >> ch;
    ch = tolower(ch);
    cin.ignore(100, '\n');
    return ch;
}
```

class CardCollection

As usual, constructor for class `CardCollection` should initialize its data members. It is not absolutely necessary to initialize all members. For example, we know that there is no possibility that the `fFileName` field would be used before being set, so it is acceptable to leave this uninitialized. On the whole, you should be cautious and initialize everything!

constructor

```
CardCollection::CardCollection()
{
    fFindString = NULL;
    fFindPos = -1;
    fNumCards = 0;
}
```

Opening and closing the associated file

Function `Open()` really consists of two parts. The first deals with the case of an existing file; this uses the auxiliary `Load()` function to get the data. The second part deals with the case where it is necessary to create a new file. Function `Close()` needs a call to `Save()`, the actual file closing action, and some tidying up operations. (The tidying up isn't comprehensive; we don't get rid of the array of pointers owned by the `DynamicArray`. The next chapter covers "destructor" – special automatically invoked tidy up routines. Class `DynamicArray` really needs a "destructor" function to tidy away its pointer array.)

The initialization step of class `UserInteraction` also involves a call to a `CardCollection::ReportStatus()` function. This function is not shown. It would simply print out the name of the file associated with the `CardCollection` and the number of cards that it contained.

```
int CardCollection::Open(const char filename[])
{
    /* Keep copy of file name */
    fFileName = new char[strlen(filename)+1];
    strcpy(fFileName, filename);
```

```
    fFile.open(fFileName, ios::in | ios::out | ios::nocreate);
    if(fFile.good()) {
           Load();
           return 0;
           }

    fFile.open(fFileName, ios::in | ios::out);
    if(!fFile.good())
           return -1;
    return 0;
}

void CardCollection::Close()
{
    Save();
    fFile.close();
    // Should get rid of data structures like the cards
    for(int i = fNumCards; i > 0; i--) {
           RefCard *r = (RefCard*) fStore.Remove(i);
           delete r;
           }
    if(fFindString != NULL) delete [] fFindString;
}
```

Reading the cards

The `Load()` function reads the number of cards then loops creating cards and getting them to read their data. Each card gets "appended" to the dynamic array. You would need some error checking on input operations even if, as here, it is limited to stopping the program if something seems to have gone wrong.

The `Save()` function is not shown. It just writes details of the size of the collection, then gets each member card to write itself to the file.

```
void CardCollection::Load()
{
    fFile.seekg(0);
    fFile >> fNumCards;
    fFile.ignore(100,'\n');
    for(int i=0; i < fNumCards; i++) {
           RefCard *r = new RefCard;
           r->ReadFrom(fFile);
           if(!fFile.good()) {
                  cout << "Sorry, file must be corrupt, "
                                "giving up." << endl;
                  exit(1);
                  }
           fStore.Append(r);
           }
}
```

AddCard()

The function `AddCard()` is a simple implementation of the ideas shown in the interaction diagram shown in Figure 22.4:

```
void CardCollection::AddCard()
{
    RefCard *r = new RefCard();
    r->GetData();
    fStore.Append(r);
    fNumCards++;
}
```

DeleteCard()

The delete operation involves removing a chosen card from the `DynamicArray fStore`, deletion of the object, and updating of the member count. (Strictly, the member count is redundant as we could always ask the `DynamicArray` for the number of items that it holds).

```
void CardCollection::DeleteCard(int cardnum)
{
    RefCard *r = (RefCard*) fStore.Remove(cardnum);
    delete r;
    fNumCards--;
}
```

ShowCard(), ChangeCard()

The `ShowCard()` and `ChangeCard()` functions are similar. A pointer to the chosen card is obtained from the `DynamicArray`. Then the card is told to perform an action. Function `ShowCard()` illustrates both:

```
void CardCollection::ShowCard(int cardnum) const
{
    RefCard *r = (RefCard*) fStore.Nth(cardnum);
    r->Show();
}
```

Identifying a field for search or display

Function `GetField()` has to prompt the user for an indication of the search field (taking into account whether "any" is an allowed response). It can return a -1 value for an illegal input of a positive integer constant identifying a valid field. The constants like `kAUTHORFIELD` will have to be defined in RefCard.h.

```
int CardCollection::GetField(int anyallowed)
{
    cout << "Which data field?" << endl;
    cout << "a Authors, t Title, c Content (abstract), "
                  "k Keywords" << endl;
    if(anyallowed)cout << "x for any of these" << endl;
    char ch;
    int result = -1;
    cin >> ch;
    switch(ch) {
```

```
case 'x': if(anyallowed) result = kANYFIELD; break;
case 'a': result = kAUTHORFIELD; break;
case 't': result = kTITLEFIELD; break;
case 'c': result = kABSTRACTFIELD; break;
case 'k': result = kKEYWORDFIELD; break;
           }
    return result;
}
```

Viewing a collection

Function `DoView()` had better check that there are some cards too view. If there are, it needs to use `GetField()` to let the user pick a field to be displayed ("any" is not allowed here). If the user enters a valid field selection, then each card in the collection gets told to print the contents of that field. (Loops like the `for(;;)` loop here run from 1 to N because the `DynamicArray` uses 1 for the first element, a departure from the normal C convention of zero-based arrays).

```
void CardCollection::DoView()
{
    if(fNumCards < 1) {
           cout << "No cards to view!" << endl;
           return;
           }

    int field = GetField(0);
    if(field < 0) {
                  cout << "Invalid field choice, ignored." << endl;
                  return;
                  }
    for(int i = 1; i <= fNumCards; i++) {
           cout << i << "\t";
           RefCard *r = (RefCard*) fStore.Nth(i);
           r->PrintField(field);
           cout << endl;
           }
}
```

Finding a particular card

Function `DoFind()` has some similarities to `DoView()`. It starts by getting the field selection ("any" is allowed); then prompts for and reads the search string. It makes a copy of this string, the copy is saved in the `CardCollection's` state data. This makes it possible to resume the search if requested.

Once all the search data are entered, the function loops getting and checking successive cards from the collection. If a match is found, the card is displayed, and the function returns. The data member `fFindPos` is used to control the loop and it retains the position where a match is found.

If there is no match in the entire collection, a warning is displayed and the search data are tidied away.

```
void CardCollection::DoFind()
```

Identifying the search field

Getting the search string

Saving a copy of the search string

Search loop

Process a successful match

Report failure and tidy up

```
{
    fSearchField = GetField(1);
    if(fSearchField < 0) {
                cout << "Invalid field choice, ignored." << endl;
                return;
                }
    char buff[50];
    cin.ignore(100,'\n');
    cout << "Enter search string : ";
    cin.getline(buff,49, '\n');

    if(fFindString != NULL) delete [] fFindString;
    fFindString = new char[strlen(buff) + 1];
    strcpy(fFindString, buff);

    for(fFindPos = 1;fFindPos<=fNumCards; fFindPos++) {
           RefCard *r = (RefCard*) fStore.Nth(fFindPos);
           int match = r->CheckField(fSearchField, fFindString);
           if(match) {
                  r->Show();
                  return;
                  }
           }
    cout << "No Matches" << endl;
    delete [] fFindString;
    fFindString = NULL;
    fFindPos = -1;
}
```

Find again

The function `DoFindAgain()` must check that there is a search in progress and that we haven't already reached the end of the collection. If further search is meaningful, the function loops looking for the next matching card. As in `DoFind()`, a match results in display of a card and return from the routine while failure to get a match results in a warning.

The code should work even if the user does things like delete cards between successive find again operations.

```
void CardCollection::DoFindAgain()
{
    if(fFindPos < 0) {
           cout << "You've got to do a 'Find' before "
                        "'Find Again'" << endl;
           return;
           }
    if(fFindPos >= fNumCards) {
           cout << "No more matches" << endl;
           delete [] fFindString;
           fFindString = NULL;
           fFindPos = -1;
           return;
```

```
            }

    for(fFindPos++; fFindPos<= fNumCards; fFindPos++) {
            RefCard *r = (RefCard*) fStore.Nth(fFindPos);
            int match = r->CheckField(fSearchField, fFindString);
            if(match) {
                    r->Show();
                    return;
                    }
            }
    cout << "No more matches" << endl;
    delete [] fFindString;
    fFindString = NULL;
    fFindPos = -1;

}
```

class RefCard

The constructor should initialize the strings to null and the numeric fields to zero:

```
RefCard::RefCard()
{
    fAuthors[0] = '\0';
    fTitle[0] = '\0';
    fJournal[0] = '\0';
    fKeywords[0] = '\0';
    fAbstract[0] = '\0';
    fFirstPage = fLastPage = fIssue = fYear = 0;
}
```

File transfer

The functions for transfer to and from file are simple. The file used here is a text file rather than a binary file. It should be possible to read and edit such a file using a word processor. (You may find that you have to use one of the utility programs on your system to change the "file type" before you can edit these files. In some IDEs, data files written by programs are created with non-standard types.)

The final call to `ignore()` in `ReadFrom` is there to consume the newline after the last of the numbers. If you don't consume this newline, the next `ReadFrom()` operation will fail as it will encounter the newline character when trying to read the "authors" string and so get out of phase.

```
void RefCard::WriteTo(fstream& s) const
{
    s << fAuthors << endl;
    s << fTitle << endl;
    s << fJournal << endl;
    s << fKeywords << endl;
    s << fAbstract << endl;
    s << fFirstPage << " " << fLastPage << endl;
    s << fIssue << " " << fYear << endl;
}
```

```
void RefCard::ReadFrom(fstream& s)
{
    s.getline(fAuthors, kNAMEFIELDSIZE-1,'\n');
    s.getline(fTitle, kNAMEFIELDSIZE-1,'\n');
    …
    s >> fFirstPage >> fLastPage >> fIssue >> fYear;
    s.ignore(100,'\n');
}
```

GetData()

Function `GetData()` is similar to `ReadFrom()` except that it prompts for each data item before reading:

```
void RefCard::GetData()
{
    cout << "Enter data for paper:" << endl;

    cout << "Author(s) : ";
    cin.getline(fAuthors, kNAMEFIELDSIZE-1, '\n');
    …
    …
    cout << "Enter page range,\n";
    fFirstPage = GetNumber("\tfirst page: ");
    fLastPage = GetNumber("\tlast page: ");
    fIssue = GetNumber("Issue # : ");
    fYear = GetNumber("Year : ");
}
```

Numeric values are required for the page numbers, year etc. We need code that prompts for a number, and then checks that it gets a (positive non zero) number. If the user enters something that is not a number, we have to clear the error condition, remove all characters from the input buffer and prompt again. Obviously, this code should be as a subroutine, we don't want the code duplicated for each numeric field. Hence, we have a `GetNumber()` routine.

This could be made a member function of `RefCard` but it doesn't really seem to belong. Instead it can be a filescope function defined in the RefCard.cp file:

Auxiliary, non-member function GetNumber

Output prompt and read value

If bad input, clear flag and buffer

```
static int GetNumber(char *prompt)
{
    int val = 0;
    int ok = 0;
    while(!ok) {
           cout << prompt;
           cin >> val;
           if(cin.good()) ok = 1;
           else {
                  cout << "??" << endl;
                  cin.clear();
                  cin.ignore(100,'\n');
```

```
                }
            }
    return val;
}
```

Function `Show()` just outputs field labels and values: ***Show()***

```
void RefCard::Show() const
{
    cout << "Author(s)\t: " << fAuthors << endl;
    cout << "Title\t\t: " << fTitle << endl;
    …
    cout << "Abstract\t: " << fAbstract << endl;
}
```

Function `Change()` has to prompt for a field identifier, then it should display the contents of the field before reading a new value: ***Change()***

```
void RefCard::Change()
{
    cout << "Changing card:" << endl;
    cout << "Select a (Authors), t (Title), j (Journal)" << endl;
    cout << "\tk (Keywords), c (Content, abstract)" << endl;
    cout << "\ty (Year), v (Volume), p (Page range)" << endl;
    char command;
    cin >> command;
    command = tolower(command);
    cin.ignore(100,'\n');
    switch(command) {
case 'a':
            cout << "Authors currently " << fAuthors << endl;
            cout << "Enter correction : ";
            cin.getline(fAuthors, kNAMEFIELDSIZE-1, '\n');
            break;
    …
    …
case 'v':
            fIssue = GetNumber("Volume");
default:
            cout << "??" << endl;
            }
}
```

Function `PrintField()` simply outputs the contents of a chosen field: ***PrintField()***

```
void RefCard::PrintField(int fieldnum)
{
    switch(fieldnum) {
case kAUTHORFIELD: cout << fAuthors;  break;
case kTITLEFIELD: cout << fTitle;  break;
```

```
case kKEYWORDFIELD: cout << fKeywords;  break;
case kABSTRACTFIELD: cout << fAbstract; break;
            }
}
```

While function `CheckField()` uses `strstr()` to check whether a given string is contained in any of the string data fields:

CheckField()

```
int RefCard::CheckField(int fieldnum, char *content)
{
    if((fieldnum == kANYFIELD) || (fieldnum == kAUTHORFIELD))
            return (NULL != strstr(fAuthors, content));
    if((fieldnum == kANYFIELD) || (fieldnum == kTITLEFIELD))
            return (NULL != strstr(fTitle, content));
    …
    …
    return 0;
}
```

Function `strstr(const char *s1, const char *s2)` "looks for a substring within a string" returning a `char*` pointer to the first place where substring `s2` can be found in `s1` (or `NULL` if it doesn't occur). If is one of the standard functions in the string library. You can find more details using your IDE's help system (or the separate "ThinkReference" program for the Symantec IDE).

22.2 INFOSTORE

Specification

The InfoStore program is to allow a user to maintain collections of news articles. These articles are to be indexed according to the "concepts" that they contain. Each concept can be represented by an arbitrary number of keywords. The vocabulary of concepts and keywords is to be user definable; there can be a fixed maximum on the number of concepts allowed (at least a few hundred). (The concept-keyword scheme is the same as in the example in Section 18.3. For example you might have the concept "ape" matched by any of a set of keywords that includes "ape", "apes", "chimps", "chimpanzee", "bobo", "gorilla", ….)

The "InfoStore" program is to allow a user to:

- create an initial vocabulary of keywords and concepts;
- add keywords and concepts to an existing vocabulary;
- add the text of news article to the system;

- perform a search for articles with search requirements specified by entry of required and prohibited keywords.

22.2.1 Initial design outline for InfoStore

Preliminaries

For a program like this, there are a few design decisions that come before the stage where we start thinking about the objects that might be present.

Files for permanent data

We have to decide how to store the permanent data in disk files. We now have three different kinds of data.

Data files

There are the actual news articles. These are just blocks of text and, as in the program in Section 18.3, we can store them all in a single file provided we can separate them (use null, '\0', characters) and we know where each begins.

Index files

Next, there are going to be index entries, one for each article in the main articles file. As in the example in Section 18.3, these are going to consist of a set of bits (bit value 1 implies presence of a particular concept in an article, value 0 implies absence), and a "file address" (record of where an article starts in the main file). We will need a second file to hold these index entries.

Vocabulary files

Finally, we need to have some form of "vocabulary file". The earlier program used a fixed "compiled-in" vocabulary; but that is too restrictive. We now need a file that contains keywords and numbers. The number associated with a keyword will identify the concept to which it belongs. So, if for example, concept 25 is the system's representation of "ape", the vocabulary file should contain entries including "ape 25", "chimpanzee 25", "gorilla 25". (It is assumed that keywords cannot be associated with multiple concepts; so you can't have concept 25 "ape" and concept 95 "gangster" with gorilla appearing as both "gorilla 25" and "gorilla 95".) The vocabulary file can be a simple text file with lines that contain keyword concept number pairs. The program can build any more elaborate vocabulary structures from this input.

Common "base name" for files

Each "information store" thus needs three files. We can keep them together by allowing the user to define a "basename" for the files, e.g. "travel" or "science", and create the files with distinguishing suffixes (e.g. "travel.dat", "travel.ndx", and "travel.vcb").

Affect of changing the vocabulary

If the user extends the vocabulary then any articles already stored will have to be re-indexed. The program should deal with this automatically.

Memory resident data and disk-based data

The program will load all the vocabulary data and build a look-up structure that can be used to check whether a word corresponds to one of the concepts. The articles and index entries will be left in the files. When a search is being performed, each index entry in turn will get loaded into memory; the user's query can be represented by a Bitmap structure similar to that used an index entry. Articles that match a search request will just be copied character by character from the data file to the output; they don't have to be loaded completely into memory. When an article is being added, it gets copied character by character from input file to the data file. Again, the complete article never has to be in memory. While adding an article, the program will build up a new index entry and store, temporarily, the current word from the article as was done in the program in Section 18.3

Thus, the only large memory resident structures will be those used for the vocabulary.

Program operation

Normally, the user would be expected to start by building up a vocabulary. This would probably be done in several separate runs of the program with each run adding a few more keywords. Once a basic vocabulary had been established, articles might start to be entered. The user would want to identify a text file with an article. The system should copy the contents of this file to its data file, at the same time building up an index record that would then get written to the index file. In any particular run of the program, the user might add a few articles, and make one or two searches.

Finding objects

Once you have resolved preliminaries like how the files might be organized and how the program would be used, you can start postulating possible objects.

"Obvious objects"

There are a few "obvious" objects. As in the last example, the system will probably use a "UserInteraction" object that organizes most of the processing. Once again, the main program will do little more than create this `UserInteraction` object and tell it to "run".

UserInteraction object

The `UserInteraction` object will accept commands like "expand vocabulary", "add an article", "do a search". The `UserInteraction` object might need to get some additional information but would deal with most requests by passing them on to other objects in the system.

Vocabulary object

Another plausible candidate is a "Vocabulary" object. Something has to own the keywords and concept numbers. This something has to have organized some fast lookup mechanism (hash table or tree). This something has to get the words in from the vocabulary file, and back out to the file whenever extra words have been added during a run of the program. Words have got to be looked up and concept numbers returned. There certainly seems to be a group of related data, and larger number of operations on these data, that could be packaged up in an object that is an instance of some "Vocabulary" class.

InfoStore object

Another possible candidate is an "Infostore" object. Once again, "something" has to own the three files, and the string that represents the basename of the files. This "something" can organize opening of the files, keeping track of the length of the data files and the number of entries in the index file. It can forward user requests for changes to the vocabulary on to the vocabulary object and perform other organization roles.

Words?

There are several other possible objects. Maybe "Words" should be objects. A Word object could own a string and a concept number. But there don't seem to be many tasks for a Word to perform. The Vocabulary object might employ Words, but the rest of the program would almost certainly just be working with character strings. Words can be left for now. They might reappear later on but they don't seem to have sufficient of a role to justify consideration during preliminary design.

News Articles?

News articles? No. They certainly don't do anything (other things scramble through the text of a news article). Although articles may exist as an "objects" in the data file, they never really exist in memory. Most processing involving articles works on them one character at a time.

ArticleFinder?

How about an "ArticleFinder" object? You could argue that this "owns data" e.g. it will own the user's query (this will take the form of a set of required and set of prohibited concept numbers, both represented as bit maps). It will check this query against each entry in the index file and print those articles that match.

Don't go confusing a function (verb) with an object (noun)!

However, an ArticleFinder object doesn't really have the right feel. Objects are primarily things that are created dynamically and remain around for a reasonable length of time (like the RefCards in the last example). If you think how the program would work, the InfoStore object would create an ArticleFinder each time the user made a search request. This ArticleFinder would do its stuff. Then it would be destroyed. It doesn't have the right kind of lifetime. It really is "just a function" and the data that it "owns" are really just automatic variables that it uses. So discount ArticleFinder – not an object. The InfoStore can take on responsibility for finding matching articles; it will just use a group of member functions to do this.

Lists and collection classes

Lists? Use as needed. The program will hold in memory just single index records, single queries. Most of the data are left in the files. The only large collection will be the words of the vocabulary. We can add a list (or dynamic array) if needed.

Bitmap

Bitmap? Yes. We can reuse class `Bitmap` from Chapter 19. An index entry will contain a bitmap.

IndexEntry

IndexEntry? Plausible. If we define an IndexEntry class we have a place to put related behaviours like checking for matches with other `Bitmaps` that represent queries. An IndexEntry will own a `Bitmap` and a long integer to record the location of an article. It can read itself and write itself to file. It can be built up by telling it to set concept bits. Class `IndexEntry` seems to earn its way.

HashTable?

The "Vocabulary" object will create some hash table data structure. But this probably would not exist as an independently defined class. If you had a class library with a "reusable HashTable" you might proceed differently. However, reusable HashTables are not common. HashTable structures tend to be purpose built for specific applications with minor variations to adapt to special needs.

The most plausible classes are illustrated in Figure 22.9. As in the RefCards example, most are "fuzzy blob" classes because we haven't really defined what any do or own. This time, class `Bitmap` is the exception. Once again, because it is a known reusable class it can be shown with firmly defined boundaries.

In this example, the preliminary classes shown in Figure 22.9 did become the final classes used in the implementation. However, it would not be unusual for changes to be made during the later more detailed design steps.

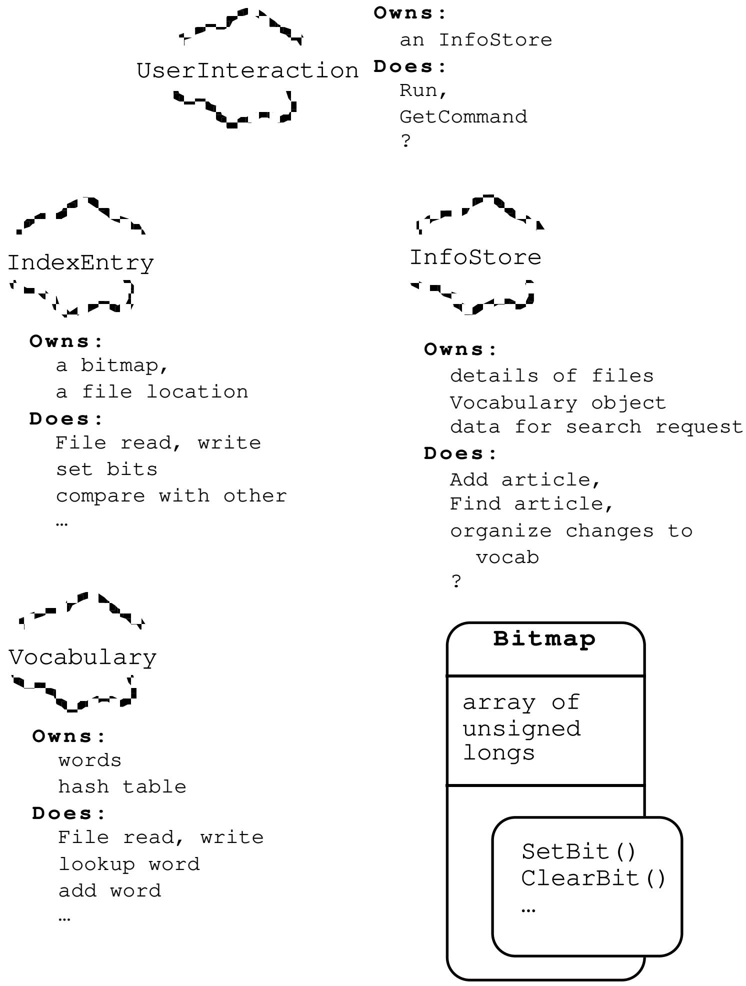

Figure 22.9 First idea for classes for InfoStore example.

22.2.2 Design and Implementation of the Vocabulary class

As in the previous example, our next task is to elaborate these initial ideas of classes. Once again, this will involve using scenarios to examine possible interactions among objects. Although the processing steps involved may be more elaborate, the actual patterns of interaction are more limited in this example.

This example does illustrate another aspect of the use of objects. An object-based approach often makes it possible to design and implement parts of a program in total isolation. Once the parts have been made to work, you fit them together to make a whole. It is just the same process as we have been doing with "reusable classes" like `DynamicArray`, except that the classes developed will only be "used" in a test context and then "reused" in the final program product.

Separate development of parts is of great practical importance. Most programs are built by teams. It is obviously more practical for individual team members to work on clearly separate parts. Separate development helps even if a single programmer is developing the system. Separate development means that the programmer is writing, and testing, two or more simple programs rather than one larger more complex program.

Focus on isolable Vocabulary object

The "Vocabulary" object represents a fairly substantial part of the program. It owns quite a lot of data in varied forms. It is certainly going to own the actual vocabulary entry items (concept number and keyword string) and a hashtable structure allowing fast lookup. It may own other data. It is going to have to do things like add words to its collection and lookup words to see if they are already in the collection. However, it doesn't need services of other objects and probably it is only the `InfoStore` object that ever requests actions by the `Vocabulary` object. Thus, it is a good candidate for separate development.

We have to start by examining scenarios that focus on use of the `Vocabulary` object. Together these will define the "public interface" for a `Vocabulary` class. Once this has been defined, separate development is possible. The design and implementation of the `Vocabulary` class can be completed and verified using a little test program that exploits the same public interface.

Scenarios for file input and output

So, what does a `Vocabulary` object (or, more briefly, a `Vocab` object) get asked to do? Firstly, there will be file input and output. Figures 22.10 and 22.11 illustrate plausible scenarios. Activity will start with the user telling the `UserInteraction` object to "open" an `InfoStore`. This will result in an "open store" request being passed to the `InfoStore` object. We can ignore most of the activity of `InfoStore::OpenStore()` for now; it will involve getting a "base name" from the user and then opening of all three files. (Inconsistencies such as only one or two files existing will terminate the program.) The scenario in Figure 22.10 picks up at the point where the files have all been opened successfully. The `InfoStore` object will ask the `Vocabulary` object to load its data from the already opened vocab-file.

The first item in the file might as well be an integer giving the number of words in the vocabulary. There could be a few thousand words. We want to allow for a few hundred concepts and each concept might be represented by several different keywords in the news articles. So we can expect hundreds, possibly thousands, of keyword/concept number pairs.

VocabItem struct

We will need something to store these data. As noted earlier, we might use instances of some class `Word`. But at least for the present it appears that we could make do with a simple struct like the `VocabItem` used in the earlier simpler version of the program.

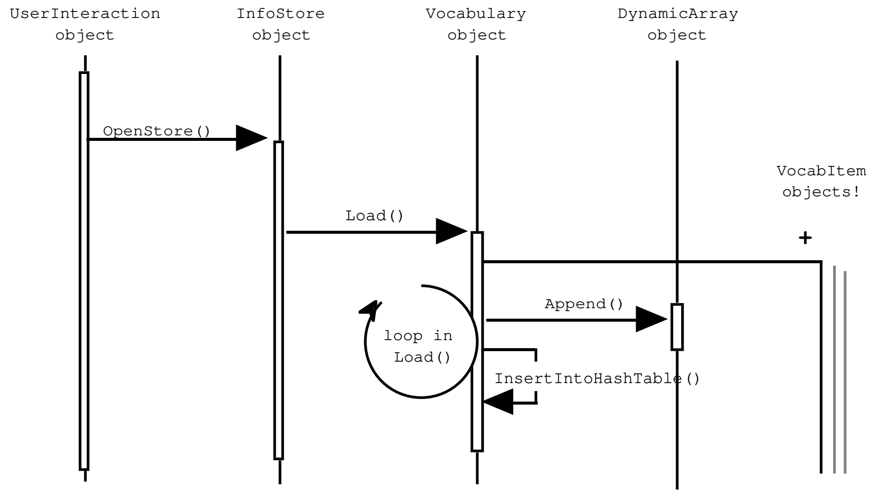

Figure 22.10 Object interactions while loading a vocabulary file.

VocabItems

A `VocabItem` struct will have a `char*` pointer and an integer. Function `Vocabulary::Load()` can have a loop in which it creates `VocabItems`. If these are just simple structures, then the `Vocab` object better do the work of reading their data. It reads a word into a temporary buffer. A new character array can be created (this operation is not shown in Figure 22.10) and the word gets copied into the new array. The character array's address can be stored in the new `VocabItem` along with the integer concept number also read from file.

A dynamic array to store the VocabItems

We have to store the complete collection of words. We will have to provide a service like "list all the words with their concept numbers", and "list all words associated with concept number …". So, we will be working sequentially through the collection.

For this collection we can obviously use an instance of class `DynamicArray`; it can be a data member in the `Vocab` object. Once a `VocabItem` has been created and its data fields filled in, it can be added to the dynamic array. This is shown in Figure 22.10.

Separate hash array

We also need the fast lookup version. This will be a hash table of pointers to the same `VocabItems`. The array used for hash address will have to be larger than the maximum number of keywords we expect. We had better arrange to have it created in the constructor for class `Vocab`. Once we have read a `VocabItem`, we have to add it to the hash table in addition to the main dynamic array. In Figure 22.10, this is illustrated as the call to `InsertIntoHashTable()`. Obviously, this is a non-trivial process. Later it will get broken down using a "top down functional decomposition" approach. This more detailed design step will add some other private member functions to class `Vocab`.

Figure 22.11 illustrates those parts of a `Close()` operation that involve the `Vocab` object. It will loop, getting the `VocabItems` from its array and writing their contents to file. It would be worthwhile checking whether the vocabulary has been changed; there is no need to spend time rewriting the file if the existing file is still valid. Class `Vocab` should have some boolean or integer indicator, `fChanged`, that gets set when words are added. If this is not set, the write to file step can be omitted. It is possible that another lot of data might get loaded into the same `Vocab` object, so once the current data are finished with the arrays should be cleared and the existing `VocabItems` should be deleted.

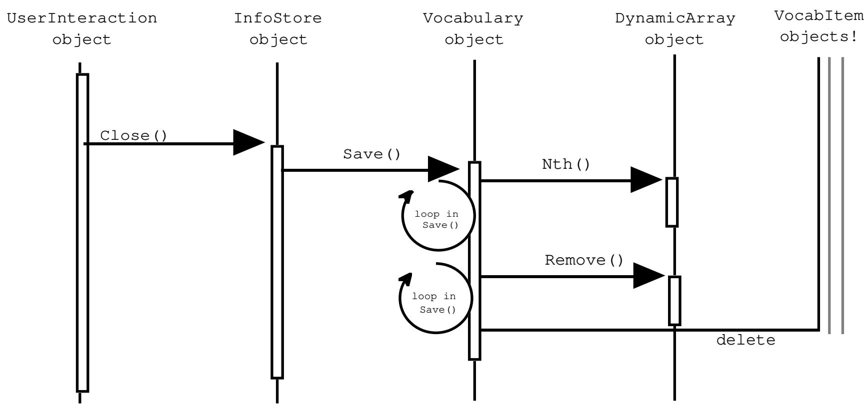

Figure 22.11 Object interactions while saving to a vocabulary file.

So far, we seem to have:

```
class Vocab {
public:
    Vocab(?);
    /*
    File support
    */
    void   Load(fstream& in);
    void   Save(fstream& out);
    /*
    Status
    */
    void   ReportStatus() const;

    …
private:
    struct VocabItem {
           short fCNum;
           char* fWord;
           …      // Maybe other data
           };
    void   InsertIntoHashTable(VocabItem*);
```

```
        int     fNumWords;
        int     fNumConcepts;
        int     fChanged;

        VocabItem **fHashTable;

        DynamicArray fTbl;
        …
    };
```

Here it has been assumed that `VocabItem` is essentially a private struct used only by class `Vocab`. It would be useful for the `Vocab` object to maintain counts of the number of keywords and concepts that it had defined; hence data members like `fNumWords`. A `ReportStatus()` member function might be useful. It could print out details of the number of keywords and the number of concepts.

The hash table is an array of pointers to `VocabItems`. Since it is a dynamically allocated structure located somewhere in the heap, the type of `fHashTable` is pointer to (an array of) pointer(s) to `VocabItems`, or `VocabItem**`. (This is as discussed previously in Chapters 20 and 21.)

What else might a Vocab object do?

Now that we can load a `Vocab` object from a disk file, what should we do with it?

The `UserInteraction` object will offer a basic menu of commands like "do search", "do addition of article", and "do vocabulary operations". These will results in requests to the `InfoStore` object to do the search, or addition, or organize vocabulary options. In the case of vocabulary modifications, the `InfoStore` object will probably present the user with a kind of submenu. The commands will be things like:

InfoStore command options involving a Vocab object

- Concept
  Add new concept by giving the first keyword. (The `Vocab` object should allocate a new concept number).

- Word
  Add another word for an existing concept. The user would then have to enter the concept number and the new keyword (system should check the number is in range).

- List
  List all concepts or words. There would have to be another prompt to find whether the user want a printout showing all words, or just those associated with a particular concept number (or, maybe, a list arranged by concept number).

- Test
  Test whether word is associated with a concept. This is really just a "lookup" operation on a word provided by the user.

The keyword/concept-number idea is not very well defined. Really, the only difference between adding a concept and adding an alternative keyword is that for a "concept" the

`Vocab` object selects the next possible concept number whereas for a keyword the user has to specify an existing concept number.

The `Vocab` object could check that the user doesn't request a new concept or keyword and then enter a word that already exists. However, the user can do this check anyway by using the Test option to check a word before trying to enter it.

These different tasks involve three basic patterns of interactions among the objects. None seem sufficiently elaborate to merit an object interaction diagram.

Creational patterns for VocabItems

The first pattern is a "creational pattern" which will get used for the Concept and Word commands. The `InfoStore` object gets the necessary data from the user (a text string, and in the case of the Word command the input will also contain a concept number). The `InfoStore` invokes member functions `Vocab::AddNewConcept()` or `Vocab:: AddExtraWord()`. These functions build a new `VocabItem`, and then add it to both the hash table and the dynamic array. The functions had better return integer error codes. There will be a limit on the number of concepts; there may be other constraints that could cause these operations to fail.

The `InfoStore` object had better check the validity of any concept number entered with an extra keyword. It can pass the number entered by the user to the `Vocab` object to check; this could be done before any call to `AddExtraWord()`. So class `Vocab` had better provide a `CheckConceptNum()` member function.

The additional parts of class `Vocab`'s public interface identified by considering these interactions are:

```
int         Vocab::CheckConceptNum(int conceptnum) const;
int         Vocab::AddExtraWord(const char aWord[], int conceptnum);
int         Vocab::AddNewConcept(const char firstWord[]);
```

Iterating through the dynamic array

The various suboptions under List would all be handled by the `InfoStore` object asking the `Vocab` object to perform a specialized listing operation. The `Vocab` object would work using a loop that looks at successive `VocabItems` from its dynamic array and prints the appropriate ones. Listing all words is easy; the loop in a `ListWords()` function simply prints every `VocabItem`. A list of all `VocabItems` associated with a given concept number requires only an extra test and the same basic loop structure; this can be handled using a `Vocab::ListConcept(int conceptnum)` member function. Listing the keywords for each concept in turn could be handled by a function, `ListAllConcepts()`, that has a loop working through all concept numbers calling the `ListConceptNum()` function for successive numbers. Of course this means running through the array many times. This may be a bit costly, but there is no need to look for more efficient schemes, like sorting by concept number, because the execute-time is going to be determined almost entirely by the printing processes. More elaborate schemes would just add code but not produce any noticeable change in performance.

The "listing" options require the following additional functions:

```
void        Vocab::ListConcept(int conceptnum) const;
void        Vocab::ListAllConcepts() const;
```

```
void        Vocab::ListWords() const;
```

Looking up a word in the hashtable

The Test command will require that the `Vocab` object identify the concept number associated with a given keyword. It will need a function like:

```
void        Vocab::IdentifyConcept(const char aWord[]) const;
```

This will print details of the concept number, or report that the keyword is not known.

Other interactions involving Vocab object

The `Vocab` object will also be used when generating index records for new articles and creating queries. Probably, both these requests will come from the `InfoStore` object. They require the same function, it will be given the word to look up, and will return an integer concept number (a code like -1 could be used to indicate that the word is not defined).

```
int         Vocab::Lookup(const char aWord[]) const;
```

Of course, at least one `Vocab` object gets created so a constructor had better be defined. It would probably be useful if the program could specify a default size for the vocabulary. Other arguments for the constructor might be identified later.

Completing the public interface

If all the interactions involving `Vocab` objects have been identified, then we have completely characterised the public interface for the class.

```
class Vocab {
public:
    Vocab(int VocabSize);
    /*
    File support
    */
    void   Load(fstream& in);
    void   Save(fstream& out);
    /*
    Status
    */
    void   ReportStatus() const;
    int    CheckConceptNum(int conceptnum) const;

    /*
    Checking and adding words
    */
    int    Lookup(const char aWord[]) const;
    int    AddExtraWord(const char aWord[], int conceptnum);
    int    AddNewConcept(const char firstWord[]);
    /*
    Getting info on concepts
    */
    void   ListConcept(int conceptnum) const;
    void   ListAllConcepts() const;
    void   IdentifyConcept(const char aWord[]) const;
```

```
    void   ListWords() const;
private:
    …
};
```

It is now possible to complete the design, implementation and testing of this class. There is no need to build an `InfoStore` class. A simple interactive test program can easily be written to exercise the various member routines.

Detailed design, implementation, and test of class Vocab

What remains?

The remaining design work will involve minor choices on detailed representation of the data and probably some further functional decomposition for the more elaborate member functions.

Concept numbers

The index entries for the file are to use class `Bitmap`. This allows chosen bits to be tested. It stores 512 bits, numbered 0…511. The number of concepts should be limited to 512. Internally, concept numbers should be represented by integers in the range 0…511 but it would probably be best if the user saw these as 1…512 (this means that there will have to be conversions on input and output).

Array sizes

The argument for class `Vocab`'s constructor can be used to define the initial size for the dynamic array (this will involve a minor C++ feature not previously illustrated). By default, the dynamic array only grows by 5 elements; that will be too small, a larger increment should be defined, maybe 25% of the initial size.

We don't want the hash table becoming full. It will probably be worthwhile defining a maximum size for the vocabulary (some multiple of the initial size) and refusing to add words once this size is reached. If the hash table is made slightly larger, we can guarantee that it never becomes full. The entries in the hashtable will either be `NULL` or pointers to `VocabItems` created in the heap and also referenced from the main dynamic array.

Most of the functions should be straightforward. The listing functions just involve loops accessing successive `VocabItems` in the dynamic array. The hashtable functions (`Test()`, `Lookup()`, `AddExtraWord()`, and `AddNewConcept()`) will use code similar to that illustrated earlier in Chapters 18 and 20.

Another private member function

Hash keys are going to have to be computed for the various strings. The code will be the same as that illustrated previously (Section 18.2.1) but it should now take the form of a private member function for class `Vocab`:

```
unsigned long Vocab::HashString(const char str[]) const;
```

Although a good hashing function, it is relatively expensive because of its loop through all the characters in a string. It might be worth saving the hash keys in the `VocabItems`.

This would avoid the need to recompute the keys for all the words when the files are reloaded. Consequently, we might redefine `VocabItem` as follows:

```
struct Vocab::VocabItem {
    unsigned long fKey;
    short fCNum;
    char* fWord;
};
```

Implementation

Constructor

The constructor has the following form

```
Vocab::Vocab(int vocabsize) : fTbl(vocabsize, vocabsize/4)
{
    fNumWords = fNumConcepts = fChanged = 0;
    fTblSize = 5*vocabsize;
    fMaxWords = 4*vocabsize;
    fHashTable = new VocabItem* [fTblSize];
    for(int i=0;i < fTblSize; i++)
           fHashTable[i] = NULL;
}
```

The bit in bold illustrates the extra feature of C++ – initialization of data members that are instances of classes with their own constructors.

Data members that are instances of other classes

We have already had classes that had data members that were instances of other classes; after all, in the RefCards example, the `CardCollection` object had a `DynamicArray`. But in the previous examples we have been able to rely on default constructors. The default constructor for a `DynamicArray` gives it ten elements, so `CardCollections` start with an array of size ten. With `Vocab` objects, we want the `DynamicArray` to start at some programmer specified size, so we can't just leave it to the default constructor.

Data members that have their own constructors

C++ allows you to pass arguments to the constructors for any data members that require such initialization. These data member constructors get executed prior to the body of the class's own constructor. They have to be specified as shown above. They are separated from the argument list of the constructor by a colon (`:`), and are listed before the opening `{` bracket of the body.

In this example, the `fTbl` data member (the `DynamicArray`) is initialized using the `DynamicArray(size, increment)` constructor.

The hash table is made 25% larger than the maximum number of words. Thus it can never be more than 80% full and so the simple linear probing mechanism will work quite satisfactorily. The element of the hash table need to be initialized to `NULL`.

File I/O

The `Load()` and `Save()` functions are as follows:

```
void Vocab::Load(fstream& in)
```

```
{
    fChanged = fNumWords = fNumConcepts = 0;
    in >> fNumWords;
    if(in.eof()) { in.clear(); return; }
    if(fNumWords > fMaxWords) {
           cout << "Problems with file.  Seems to have too many"
                        "words." << endl;
           exit(1);
           }
    in >> fNumConcepts;
    if(fNumConcepts >= kMAXCONCEPTS) {
           cout << "Bad data in file." << endl;
           exit(1);
           }

    for(int i=0; i < fNumWords; i++) {
           VocabItem *v = new VocabItem;
           if(!in.good())break;
           in >> v->fKey >> v->fCNum;
           char lword[100];
           in >> lword;
           v->fWord = new char[strlen(lword) + 1];
           strcpy(v->fWord, lword);
           fTbl.Append(v);
           InsertIntoHashTable(v);
           }
    if(!in.good()) {
           cout << "Sorry, problems reading vocab file. "
                        "Giving up" << endl;
           exit(1);
           }
}
```

Checks on file contents

Loop creating VocabItems

Add VocabItems to both arrays

The file used for the vocabulary is really a simple text file. Consequently, a user may edit it with some standard editor or word processor. The `Load()` routine has to do some checking to validate the input. The main part of `Load()` is the loop where the new `VocabItem` structs are created, their data are read in, and they are then added to both the dynamic array and the hash table.

Function `Save()` is called when the program has finished using the current data in the `Vocab` object. If changed, the updated data should be saved to file. All existing data structures have to be cleaned out.

```
void Vocab::Save(fstream& out)
{
    if(fChanged != 0) {
           out << fNumWords << " " << fNumConcepts << endl;
           for(int i=1;i<= fNumWords; i++) {
                  VocabItem *v = (VocabItem*) fTbl.Nth(i);
                  out << v->fKey << " " << v->fCNum << " " <<
                         v->fWord << endl;
```

```
                        }
                }
        for(int j =0; j < fTblSize; j++) fHashTable[j] = NULL;
        for(j = fNumWords; j> 0; j--) {
                VocabItem *v = (VocabItem*) fTbl.Remove(j);
                delete [] v->fWord;  // Get rid of the string
                delete v;            // and the structure
                }
        fChanged = fNumWords = fNumConcepts = 0;
}
```

Tidying up is hard work!

Simple access functions

Functions like `ReportStatus()` (print details of number of words and concepts), and `CheckConceptNum(int num)` (check value against `fNumConcepts`) are all trivial so their code is not shown.

Listing functions

The listing functions are generally similar, `ListConcept()` is shown here as a representative. After checking its argument, it has a loop that works through successive elements of the `DynamicArray` (request `fTbl.Nth(i)` returns the i-th element). The `VocabItem` accessed via the array is checked, and if it is associated with the required concept its string is printed. (The code assumes that the `conceptnum` argument is defined in the internal 0..N-1 form rather than the 1…N form used in communications with the user.)

```
void Vocab::ListConcept(int conceptnum) const
{
        if((conceptnum < 0) || (conceptnum >= fNumConcepts)) {
                cout << "No such concept number." << endl;
                return;
                }
        /*
        As output for user, change to user-numbering of concepts
        (1...N) rather than internal 0...N-1.
        */
        cout << "Words mapped onto concept: #" << (conceptnum+1)
                        << endl;
        for(int i = 1; i <= fNumWords; i++) {
                VocabItem *v = (VocabItem*) fTbl.Nth(i);
                if(v->fCNum  == conceptnum) cout << v->fWord << endl;
                }
}
```

Vocabulary extension

The functions that extend the vocabulary are:

```
int Vocab::AddExtraWord(const char aWord[], int conceptnum)
{
        if(fNumWords == fMaxWords)
                return 0;
        fNumWords++;
        fChanged = 1;
```

```
    VocabItem *v = new VocabItem;
    v->fCNum = conceptnum;
    v->fKey = HashString(aWord);
    v->fWord = new char[strlen(aWord) + 1];
    strcpy(v->fWord, aWord);

    fTbl.Append(v);
    InsertIntoHashTable(v);
    return 1;
}

int Vocab::AddNewConcept(const char firstWord[])
{
    if((fNumConcepts == kMAXCONCEPTS) ||
            (fNumWords == fMaxWords)) return 0;
    AddExtraWord(firstWord, fNumConcepts);
    fNumConcepts++;
    return 1;
}
```

Calculate and save hash key

If the vocabulary is not already full, `AddExtraWord()` creates a new `VocabItem`, fills it with the given data and adds it to the hash table and the dynamic array. Function `AddNewConcept()` uses `AddExtraWord()` while providing the concept number; the count of concepts is then updated. These functions mark the vocabulary as changed so that a later call to `Save()` will result in transfer to disk.

Hash functions

The code for the hash functions can be based on that in earlier examples. A representative function from this group is `InsertIntoHashTable()`. This simply reworks earlier hash table insertion code.

```
void Vocab::InsertIntoHashTable(VocabItem* v)
{
    unsigned long k = v->fKey;
    k = k % fTblSize;
    int pos = k;
    int startpos = pos;

    for(;;) {
            if(fHashTable[pos] == NULL) {
                    fHashTable[pos] = v;
                    return;
                    }
            /*
            Shouldn't get duplicates.
            Maybe should report this as an error.
            */
            if(0 == strcmp(v->fWord, fHashTable[pos]->fWord))
                    return;

            pos++;
            if(pos >= fTblSize)
```

```
                        pos -= fTblSize;
                if(pos == startpos) {
                        /*
                        OOPS! This should never happen.
                        The hash table should never become full; entry
                        of words is supposed to be restricted so max
                        80% full.
                        */
                        cout << "Error in hashing functions of Vocab."
                                << endl;
                        exit(1);
                        }
                }
}
```

Test

With class `Vocab` defined, we need a test program. This will simply be another of those small interactive program where the user is given a menu of commands like "add a word", "list concept" and so forth. Small auxiliary routines will prompt the user for any necessary data and invoke the appropriate operations on an instance of class `Vocab`.

This "scaffolding" code belongs with the `Vocab` class and should be considered as part of the class's documentation.

Part of the `main()` function of this test program is:

```
…
#include "Vocab.h"

Vocab gv(1000);

…

int main()
{
    fstream testfile("testfile", ios::in | ios::out);
    gv.Load(testfile);
    gv.ReportStatus();
    int done = 0;
    for(; ! done ; ) {
           cout << "Enter command : ";
           char ch;
           cin >> ch;
           ch = tolower(ch);
           switch(ch) {
case 'q':  done = 1; break;
case 'c':  AddConcept(); break;
case 'l':  List(); break;
case 'w':  AddWord(); break;
```

```
case 't':   TestWord(); break;
case '?':   cout << "Commands are" << endl;
            cout << "\tl List all concepts or words." << endl;
            …
            cout << "\tq Quit" << endl;
            break;
default:
            cout << "? Unrecognized command " << ch << endl;
            break;
            }
    }
    gv.Save(testfile);
    testfile.close();
    return 0;
}
```

An example of the auxiliary functions needed is:

```
void AddWord()
{
    cout << "Enter concept number for which you want an"
                "additional keyword : ";
    int n;
    cin >> n;
    /*
    convert from user 1..N representation to 0..N-1
    */
    if(!gv.CheckConceptNum(n-1)) {
            cout << "That number doesn't correspond to a "
                        "defined concept." << endl;
            return;
            }

    cout << "Enter extra keyword : ";
    char aWord[40];
    cin >> aWord;
    if(gv.AddExtraWord(aWord,n-1))
            cout << "OK, added." << endl;
    else cout << "Sorry, vocab. full, can't add." << endl;
}
```

22.2.3 Other classes in the InfoStore program

With class `Vocab` completed and tested, development of the rest of the system could resume. As illustrated in Figure 22.12, the situation has changed. Now class `Vocab` can be treated as a predefined "reusable" component.

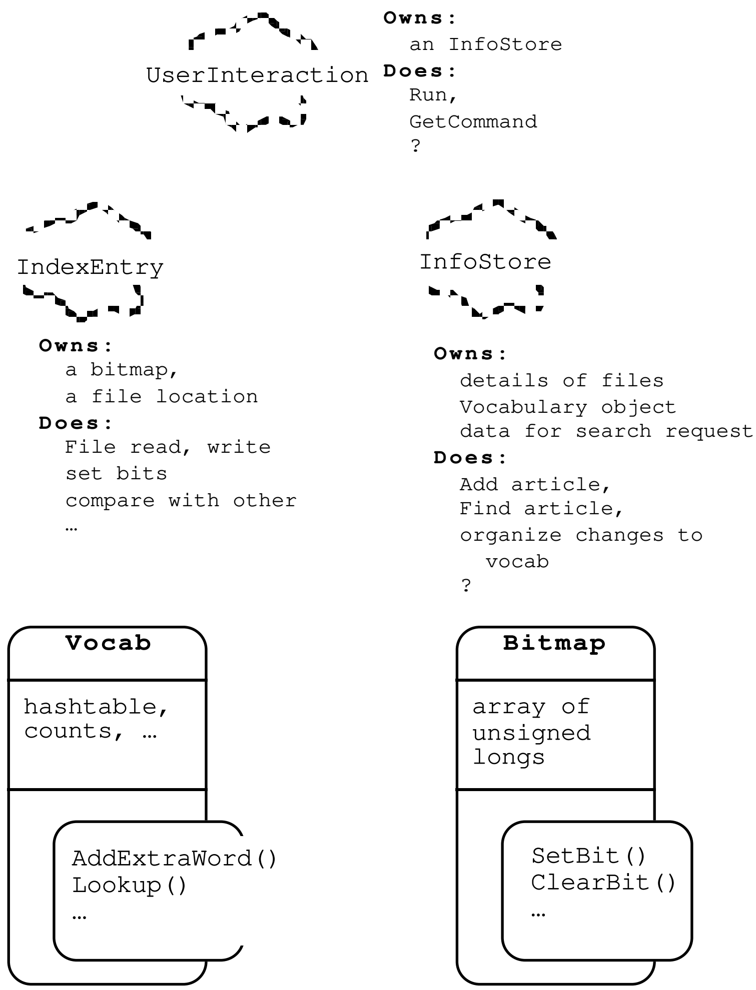

Figure 22.12 Revised model for classes.

Class UserInteraction

Class `UserInteraction` won't present many problems. This class, and the `main()` function, will be similar to the corresponding parts of the previous example. We might as well use the same main program. So we can expect class `UserInteraction` to have the same public interface and parts of the same implementation structure as last time:

```
class UserInteraction {
public:
    UserInteraction();
```

```
    void   Initialize();
    void   Run();
    void   Terminate();
private:
    void   Help();
    char   GetCommand();
    // and maybe some changed stuff!
    …
};
```

Function `Run()` will present the user with a menu of options and function `Help()` will provide some explanation of the options. There will have to be some additional private auxiliary member functions that deal with the top level commands (like "Add article") by doing some validity checks, or getting some data, and then calling an appropriate member function of the `InfoStore` object.

InfoStore data member

This time, class `UserInteraction` is going to have to own an `InfoStore` object rather than a `CardCollection` object. It doesn't much matter whether it has an `InfoStore` data member or an `InfoStore*` data member. If a pointer data member is used, as in the RefCards example, the data object can be created in the `UserInteraction` constructor. However, this time we will make it an actual `InfoStore` object.

In the RefCards example, function `UserInteraction::Initialize()` opened the data file, while `Terminate()` closed the file. You could only move from one RefCard collection to another by exiting and restarting the program. It might be worth making operations a little more general.. We could have "Open" and "Close" commands in the menu offered in `Run()`. If no set of files is open, the only commands available would be "Open" and "Quit". If a set of files is open, the commands would be "Quit", "Add article", "Search", "Change Vocabulary" and "Close". (The prompt for a command should indicate the system's state; the `UserInteraction` object should probably have a data member, `fState`, to indicate its open/closed state.)

The code for `UserInteraction::Run()` will have to be along the following lines:

```
void UserInteraction::Run()
{
    int     done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(; !done; ) {
           if(fState == 0) cout << "(closed) > ";
           else cout << "(open)   > ";
           char command = GetCommand();
           switch(command) {
case 'q' :        done = 1; break;
case '?' :        Help(); break;
case 'a' :        DoAdd(); break;
case 'c' :        DoClose(); break;
case 'o' :        DoOpen(); break;
case 's' :        DoSearch(); break;
```

```
case 'v' :         DoVocab(); break;
default :
                   cout << "Command " << command << " not"
                          "recognized" << endl;
                   }
            }
}
```

The auxiliary functions like `DoSearch()` will defer most work to the `InfoStore` object:

```
void UserInteraction::DoSearch()
{
    if(fState == 0) {
           cout << "You have to have an Information Store open"
                         << endl;
           cout << "if you want to search!" << endl;
           return;
           }
    fStore.DoSearch();
}
```

The remaining functions should all be easy to code. There may not be anything to do in `Terminate()` and `Initialize()`. They got included as a move towards a standard `UserInteraction` class. These two examples in this chapter with their similar structure and `UserInteraction` classes provide, in a very limited way, a model for the more elaborate programs that can be built with the framework class libraries introduced in Part V. In those libraries, you will find standardized classes that accept commands from a user and route these to appropriate data objects.

Classes InfoStore and IndexEntry

InfoStore

The real work in this program is done by `InfoStore` along with its helpers `Vocab` and `IndexEntry`. Class `InfoStore` has to deal with a variety of requests from the `UserInteraction` object. We know that the `Vocab` object doesn't need to make requests to the `InfoStore` object, and it is pretty unlikely that the `IndexEntry` objects will need to ask anything of the `InfoStore`. Consequently, the class's public interface will be determined entirely by the needs of the `UserInteraction` object. We can therefore sketch it in now:

```
class InfoStore {
public:
    InfoStore(int VocabSize = 1000);
    int    OpenStore();
```

```
        void    Close();

        void    ChangeVocab();
        void    AddArticle();
        void    DoSearch();
    private:
        …
        fstream       fIndexFile;
        fstream       fVocabFile;
        fstream       fDataFile;
        Vocab         fVocab;
        long          fNumArticles;
        …
    };
```

The public functions are just those called from member functions of `UserInteraction`. The private data members shown have already been identified. The `InfoStore` object is supposed to own the files, since they are used for both input and output they will be `fstream` objects. The `InfoStore` need a `Vocab` object; it seems likely that it should have a count of the number of articles in the files.

There will be many additional private auxiliary member functions. The extra member functions will get identified as the known functions, like `AddArticle()`, are developed using "top-down functional decomposition".

OpenStore()

Function `OpenStore()` has to either successfully open a set of three existing files, or if none exist it should create a new set of three files. It should terminate the program if it cannot get a complete set of files. If it is able to open existing files, then this function should have a call asking the `Vocab` object to load the `VocabItems` as previously discussed.

ChangeVocab() – already been written (more or less)!

Function `ChangeVocab()` will end up very much like the little test program written to check class `Vocab`! It will have a similar prompting function that gets user commands ("add word", "list concept", etc), and similar auxiliary functions to organize things like the listing of concepts.

Another VocabChanged flag?

The `InfoStore` object will have to have its own flag data member to indicate whether the vocabulary has been changed recently. The `Vocab` object already keeps track of whether it has been changed at all since its data were loaded, and uses this to determine whether to save its data when it gets closed. The `InfoStore` object has rather different concerns. It must prevent searches of files, or closing of files, if the index entries haven't been updated to match any changes in the vocabulary.

Responsibility for updating index entries if vocabulary changes

An `InfoStore` object should set its flag data member when it asks its `Vocab` object to add a word. It should check this flag when asked to search or close files. If its flag is set, it should first go through all the index records and articles in its files. It has to repeat the indexing process by reading each article from the `InfoStore`'s main data file, updating the index entry and rewriting the updated index entry to the index file. The process is similar to that involved in the addition of a new article as described in more detail below. Once the articles have been reindexed, the `InfoStore` object can clear its version of the "vocabulary changed" flag.

AddArticle Function `AddArticle()` will start by prompting the user for the name of a file. It must then copy the content of the article to the end the data file, at the same time building up an index entry. Each word read during this copying process must be checked; any that are keywords should cause the new index entry to be updated. The interactions involved in this process are outlined below.

DoSearch Function `DoSearch()` had better start by checking whether the file contains any articles. If the data files do contains some articles, this member function has to get the query from the user. Queries consist of three parts. First there is the set of required concepts. A loop will be used to get the user to enter required keywords (the function should warn about any words entered that aren't keywords); these will be used to build up a bitmap of required concepts. The second data item is the minimum number of concepts that must match. The third item would be another set of keywords (really concept numbers) that should not be present. Once the query has been assembled, it must be checked against `IndexEntrys` read from the index file. Matches result in display of articles. Again, the interactions are outlined in more detail below.

IndexEntry

The previous slightly simpler version of this program, in Section 18.3, used some ad hoc structures to represent index entries. Although an index entry is really a composite involving a bit map and a file location, the previous representation had an array of unsigned longs and a separate long integer data element. There was no packaging of the operations on these data, the code was scattered through the other functions. With classes, we can do better.

We can now have class `IndexEntry`. This will package the data and related functions. We know some of the things an `IndexEntry` must do. An `IndexEntry` is going to have to transfer itself to/from file; this is going to be a binary transfer as they are supposed to be represented as fixed size blocks of bits in the file. An `IndexEntry` gets built up – literally bit-by-bit. When an article is processed, the concepts it contains are identified and the `IndexEntry` is told to set the corresponding bit in its bitmap. It also has to be told to note the location of an article.

These known behaviours provide a first outline for class `IndexEntry`:

```
class IndexEntry {
public:
    IndexEntry();
    void   Load(fstream& in);
    void   Store(fstream& out);
    void   SetBit(int bitnum);
    void   SetLocation(unsigned long where);
    …
private:
```

```
        Bitmap          fBits;
        unsigned long  flocation;
};
```

The `Bitmap` data member will be an instance of the class developed in Chapter 19; those `Bitmap` objects deal with things like clearing all bits, setting individual bits, and writing bit data to the file. So a lot of an `IndexEntry`'s work can be delegated to the `Bitmap` object that it owns.

Class `IndexEntry` will have some additional responsibilities related to checking matches with search queries. These will be added later when they have been more clearly identified.

Adding or re-indexing articles

Figure 22.13 illustrates the interactions among objects when an article is added of its index entry updated to reflect changes in the vocabulary.

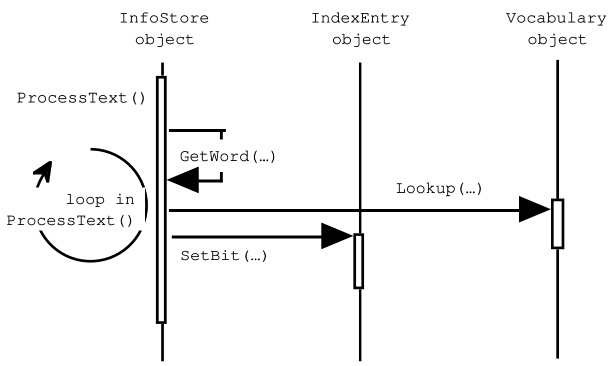

Figure 22.13 Object interactions while adding articles to information store.

Class `InfoStore` will have a `ProcessText()` private member function that deals with the detail of the indexing operations. It will have to take as arguments the input file (could be a text file with a single article or the existing data file), an `IndexEntry` to update, and a flag to indicate whether it is copying the data from the file or simply updating the `IndexEntry`. The function that calls `ProcessText()` had better set the input file so that it is at the correct position for reading (this would be the start of a new text file, but at the location of an existing article the index entries are being updated).

If it is a new article that is being added to the collection, the `IndexEntry` should already have been initialized with all its bits zero, and its `fLocation` field set to contain the current end point of the data file (the place where the copy of the new article will start). Since the vocabulary always expands, the only changes will be new words and concepts. Consequently, there is no need to reinitialize the `IndexEntry` when an article that is being re-indexed; the existing bits in the index bit map won't change, maybe a few more bits will get set.

Function `ProcessText()` will contain a loop that gets words from the file; class `InfoStore` had better have another private member function `GetWord()`.

```
while(GetWord(…) {
           int concept = fVocab.Lookup(aWord);
           if(concept>=0)
                  article_ndx.SetBit(concept);
           }
```

The `GetWord()` function will be similar to the function in Section 18.3; it will need an extra flag argument to indicate whether the characters are to be copied as well as built up into words. The "words" get filled into a character array that would be a local variable of `ProcessText()`.

Each "word" would have to be checked. Hence the call to the `Lookup()` function of the Vocab object. If `Lookup()` returns a valid concept number, the `IndexEntry` will have to be told to set this corresponding bit (the call to `SetBit()`).

The `ProcessText()` function gets executed in two circumstances. First the `InfoStore` object may have been told to add an article. The operations needed in this situation would be:

Coding a new article

```
Initialize a new IndexEntry, zeroing out its bit map
Make certain data file is positioned so that writes append data
    after all existing data
Note current end position in IndexEntry
Call ProcessText (specify data to be copied, input from new
    text file)
Write a null character to the data file to mark the end of the
    article.
Write the new index entry at the end of the index file.
Update record of number of articles.
```

Alternatively, the `InfoStore` object may be fixing up all its index records before doing a search or closing the files. The operations need in this context would be:

Fixing up existing index entries after a vocabulary change

```
for each article in collection
    Load existing index entry
    Find where related article starts
    Position data file so character read operations begin at
           articles
    Call ProcessText( specifying no copying, input from data
           file)
    rewrite updated index entry in its original position in
           the index file
```

Class `InfoStore` will need additional private member functions that organize these operations.

```
void       InfoStore::CodeArticle(fstream& infile);
void       InfoStore::FixupRecords();
```

Function `CodeArticle` would be called from the main `AddArticle()` function that opens a user-specified file with the additional news article. Function `FixupRecords()` would be called the `Close()` function (to make certain that set of index, vocabulary, and article files are consistent), and before searches. Obviously, it would start by testing the `InfoStore::fVocabChanged` flag variable to determine whether there was any need to update the files (and it would clear this flag variable once the files had been remade).

Searches

As noted earlier, the `DoSearch()` function would have to start by making certain that a search operation was valid (file contains articles, all files consistent). Then it would build up the query structure. Finally, it would have a loop that involved checking each IndexEntry from the index file against the query; articles corresponding to matching queries would be printed. The code for `DoSearch()` would be along the following lines:

InfoStore:: DoSearch()

```
if(fNumArticles == 0)
    report search not worthwhile, and return

call FixupRecords to make certain files are consistent

Build a Bitmap that represents the set of concepts required

Find minimum number of matches required

Build a second Bitmap representing unwanted concepts

Position index file at start

for each entry in file do
    load index entry

    test loaded entry against Bitmap representing excluded
           concepts, if any present then don't further check

    count number of matching concepts in index entry and
           Bitmap representing the required concepts

    if match at least the required number
           print article starting at location in index entry
```

As usual, this outline implicitly identifies a number of auxiliary functions; these will all become private member functions of class `InfoStore`.

There would have to be two auxiliary functions that build `Bitmap` objects. A `Bitmap` for a query must have at least one bit set; so a `GetQuery()` function would need to have a loop that kept prompting the user to enter "search terms"; something along the following lines:

Building a bit map that represents a query

```
Bitmap      e;
prompt for search terms
do {
    Input a word
    key = fVocab.Lookup( word );
    if(key < 0) cout << "(not used)" << endl;
    else {
           cout << "(Concept #" << key+1 << ")" << endl;
           e.SetBit(key);
           }
} while (count of concepts in query < 1
                  or user specifies another keyword);
```

As shown, the function should identify the concept numbers associated with the words entered so that the user will know whether a proposed query involves multiple concepts or whether all the keywords entered happen to map onto a single concept number. Two more simple private member functions would appear useful – something to input a word, and something to get a "yes/no" response to a prompt like "Do you want to enter another word?". When a `Bitmap` representing a query has been built, another auxiliary function can prompt the user for the minimum number of matches (it should check the query `Bitmap`, if there is only one bit set then there is no need to ask the user).

The function to get excluded words would be generally similar. The loop structure would be changed slightly because an empty `Bitmap` is valid in this context.

The extra functions need for class `InfoStore` could be:

```
Bitmap      InfoStore::GetQuery();
Bitmap      InfoStore::GetExclude();
int         InfoStore::GetRequiredNum(const Bitmap&);
void        InfoStore::InputWord(char aWord[]);
int         InfoStore::YesNo();
```

Extensions to class IndexEntry

Searches necessitate some extra member functions in class `IndexEntry`:

```
int         IndexEntry::CheckNoCommonElements(const Bitmap& bad);
int         IndexEntry::CountCommonElements(const Bitmap& good);
unsigned long      IndexEntry::Location() const;
```

The first function verifies that there are no bits in common between the `IndexEntry`'s own `Bitmap` and that given as an argument; this is used to filter out articles with excluded keywords. The second checks the number of bits that are in common. Finally, class `IndexEntry` must provide read access to the details of the location its news article. These functions are trivial to implement as all the necessary bit manipulations are provided by class `Bitmap`.

22.2.4 Final class design for the InfoStore program

The final designs for classes `InfoStore` and `IndexEntry` are shown in Figures 22.14 and 22.15.

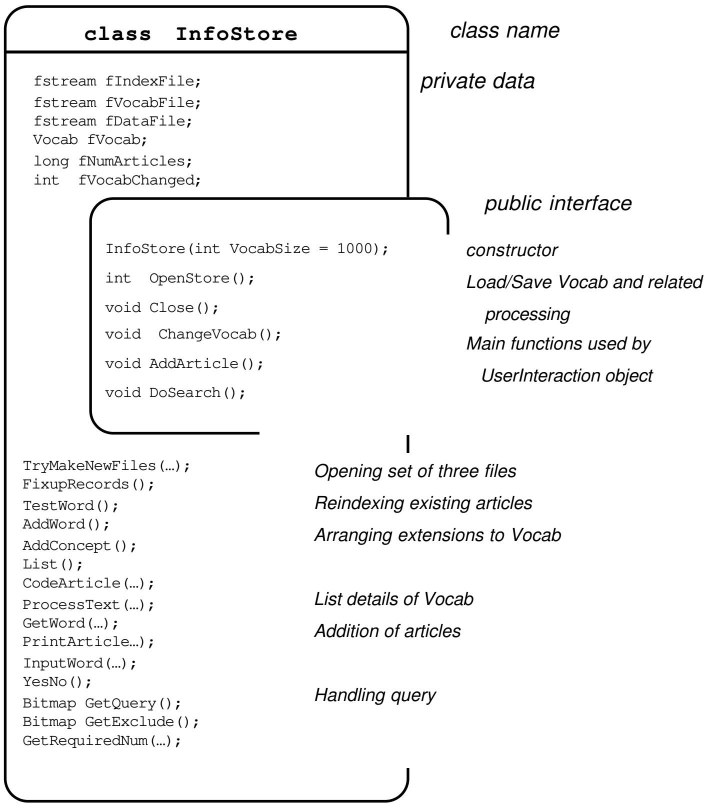

Figure 22.14 Final design for class InfoStore.

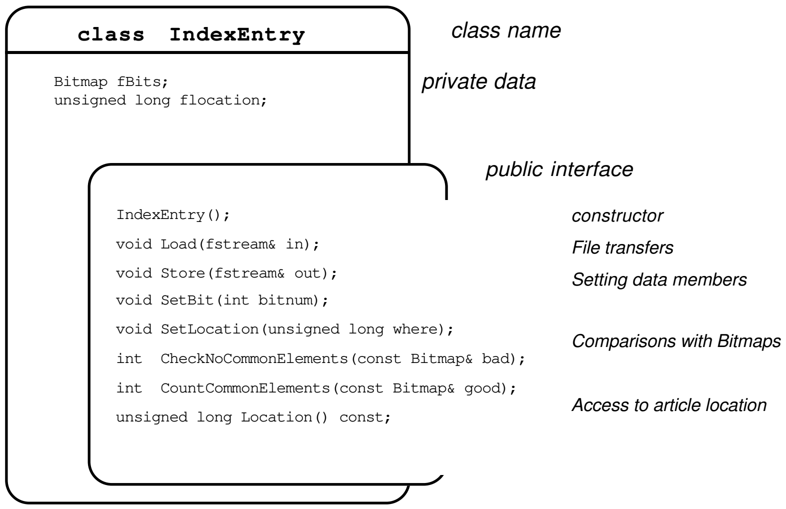

Figure 22.15 Final design for class IndexEntry.

EXERCISES

1. Complete the implementation of the InfoStore program.